Carnegie Mellon
Software Engineering Institute

Testing a Software Product Line

John D. McGregor

December 2001

20020520 225

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Testing a Software Product Line

CMU/SEI-2001-TR-022
ESC-TR-2001-022

John D. McGregor

December 2001

Product Line Systems Program

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER
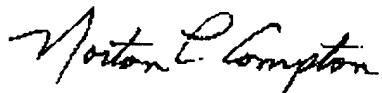
Norton L. Compton, Lt Col, USAF
SEI Joint Program Office

Table of Contents

List of Figures

List of Tables

Abstract

A suitably organized and executed test process can contribute to the success of a product line organization. Testing is used to identify defects during construction and to assure that completed products possess the qualities specified for the products. Test-related activities are organized into a test process that is designed to take advantage of the economies of scope and scale that are present in a product line organization. These activities are sequenced and scheduled so that a test activity occurs immediately following the construction activity whose output the test is intended to validate. This report expands on the testing practice area described by Clements and Northrop [Clements 02b]. Test-related activities that can be used to form the test process for a product line organization are described. Product line organizations face unique challenges in testing. This report describes techniques and activities for meeting those challenges.

1 Introduction

Software product line practice seeks to achieve a number of goals including reduced costs, improved time to market, and improved quality of the products belonging to the product line. These goals will only be achieved if quality attributes, such as correctness and reliability, are continuing objectives from the earliest phases of development. As one approach to realizing these goals, a product line organization should define a comprehensive set of activities that validate the correctness of what has been built and that verify that the correct product has been built.

Testing is one approach to validating and verifying the artifacts produced in software development. For the purposes of this report, testing will be formally defined to be

> *"The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component" [IEEE 90].*

In the context of this report, "operating" will be interpreted broadly. The software being developed is said to be operated anytime specific logical paths are followed to a particular result. This encompasses both the manual operation of any representation of the system by desk checking or inspection and the execution of the object code on a hardware platform. Proof construction based on a formal specification is an alternative to operating the software and will not be directly addressed in this report.

Testing in a product line organization encompasses activities from the validation of the initial requirements model to verification activities carried out by customers to complete the acceptance of a product. The testing activities vary in scope from encompassing the entire product line to focusing on a specific product down to examining an individual component that is one part of a product. These activities address added dimensions of complexity beyond those in a typical development project.

Testing is an essential part of developing the product line assets envisioned in the production plan. The same opportunities for large-scale reuse exist for assets created to support the testing process as for assets created for development. Since the cost of all of the test assets for a project can approach that for the development assets, savings from the reuse of test assets and savings from testing early in the development process can be just as significant as savings

from development assets. This report will discuss how testing techniques contribute to the realization of the product line's goals.

1.1 The Testing Context

Testing in the context of a product line includes testing the core assets' software, the product-specific software, and their interactions. Testing is conducted within the context of the other development activities. In this section the relationships of testing activities with other activities are identified. In Section 2, the basic testing techniques mentioned in this section are more completely defined as are the relationships between test activities.

Product line organizations build core assets and products from those assets. Core-asset building includes activities such as "Understanding Relevant Domains,"[1] "Requirements Engineering,"[1] and "Component Development."[1] Product building includes activities such as "System Integration."[1]

Product line organizations define processes that guide development in several different contexts: product line (organization wide), product, and core asset. A product line organization defines processes that guide the creation of many different types of core assets: software as well as plans, architectures, and user-oriented materials. Each process prescribes a set of activities and the sequence in which those activities are to be performed.

There are many different process models, such as the spiral, iterative, incremental, and concurrent engineering models, which can serve as the basis for processes for both core-asset creation and product building. Each has a set of assumptions that must be valid before the model is used and a set of goals that will be achieved if the assumptions are valid. There are testing techniques and strategies that are compatible with these process models.

For each specific development process, defined for a specific context and for specific product qualities, appropriate test techniques are selected. For example, performance-testing activities may not be useful for an interactive data-processing system but will be critical for an embedded system that is tightly coupled with hardware. Performance testing activities would be included in the product line process for "Architecture Evaluation"[1] and in the application engineering process for "System Integration."[1] Furthermore, if an iterative development process model is used, there is a greater need to use techniques that are easily automated.

The testing activities are related to the construction activities in a development process. A testing activity is scheduled to immediately follow the construction activity whose output the testing activity will validate. This is shown in Figure 1 as an association of each development step with a testing step. The arrows in the figure show a producer/consumer relationship. The

---

[1]   This is a software product line practice area defined by Clements and Northrop [Clements 02b].

development process produces various types of artifacts that the testing process examines. The testing process produces test results and bug reports that the development process uses to repair the artifacts. This provides the opportunity to identify defects as soon as they are injected into the artifact so that they can be removed before the faulty information is used as the basis for development in subsequent phases. Managers use this integrated process view when determining resources and schedules.

The testing activities used in a project are also related to one another as shown by the testing process thread in Figure 1. Each test activity is designed to identify the types of defects that are created in the development process phase to which it is related. An activity must also be designed to identify defects that escaped detection by the test activities earlier in the test process. These activities define a test process. Individuals with testing responsibility use this single process view when applying and interpreting test effectiveness metrics and as the focus for test process improvement.
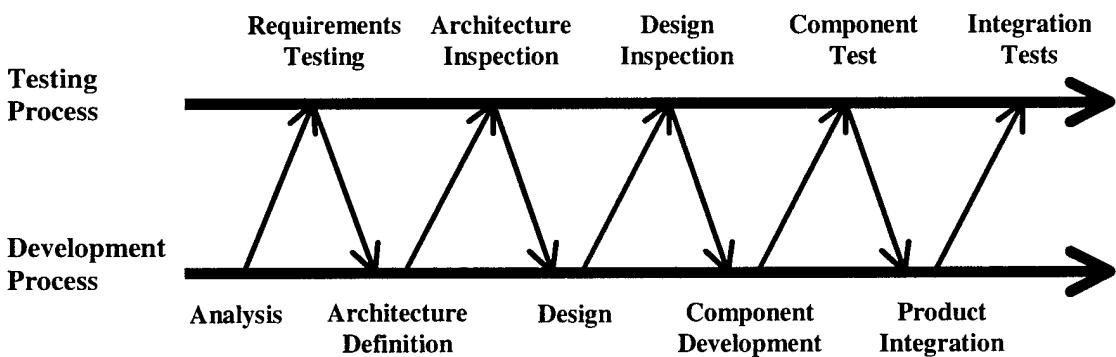
Figure 1:   Test and Development Processes

1.2 About This Report

This report describes specific practices in a software product line organization that address the validation and verification of assets and products. This report is written with the assumption that the reader is familiar with the practice areas described in the conceptual framework for software product lines developed by the Software Engineering Institute (SEI) [Clements 02b]. Testing is one of the practice areas identified there as essential to successful product line practice. The terminology used in this report will be compatible with that used in the framework.[2]

---

[2]   One additional term is added: artifact. Requirements, models, and code are artifacts until they become assets by having been tested and released for use by others.

The audience for this report is diverse. Since testing is applied at numerous points in the development process, personnel with a variety of skills are involved in the many facets of testing. System testers will find the information in Section 4 directly applicable to their work. As the project personnel are most knowledgeable about testing, they will be interested in all of the information for the purposes of guiding the project planning process. Software developers will be interested in the information in Section 5 on testing specific assets.

A project involving telecommunications equipment is used as an example in each section of this report. The example is a synthesis of experience from several projects that involve wireless devices and provides the opportunity to illustrate a number of issues including joint hardware and software development projects, "hard," real-time, embedded systems, and a rapidly evolving set of features. A brief description of the wireless device product line and the producing organization is included in Appendix A.

Section 2 presents general testing information and techniques that are common to both core assets and products. Sections 3 and 4 are divided based on types of testing activities: Section 3 addresses testing individual assets, and Section 4 addresses testing artifacts that represent complete products. Then techniques for turning artifacts into assets are discussed in Section 5. The final section presents conclusions.

2 Testing in a Product Line

This section provides basic information about testing. The basics, presented in Section 2.1, are first discussed independent of whether a product line is involved, and then issues that are specific to product lines are addressed. Underlying techniques that can be applied during core-asset development or product development are also discussed.

2.1 Testing Basics

As defined in Section 1, "testing" refers to any activity that validates and verifies through the comparison of an actual result, produced by "operating" the artifact, with the result the artifact is expected to produce, based on its specification. This may be accomplished through static testing in which an analysis of the artifact under test is conducted or through dynamic testing in which the artifact is executed and the results of the execution are evaluated. Deviations from the expected results are termed failures. A failure is considered to be the result of a defect in the artifact.

The expected results of a test are determined by an analysis of a specification. This analysis produces a tuple in which a specific stimulus is paired with the result that a correctly operating artifact is expected to produce in responding to that stimulus. The tuple is called a test case. The scope of the specification being analyzed determines the scope of the test case. For example, the specification of a class is used to produce test cases that apply to that class.

The process definition for the product line test describes the overall testing strategy for the entire product line. This process describes the relationships between the various types and levels of testing that will be performed. It prescribes roles and responsibilities in such a way that all functionality is tested, but there is no unintentional duplication. For example, the core-asset development team is responsible for testing each component based on its published specification. When a product team produces a variant of that component, that team is responsible for editing and extending that original test set to use with the new variant.

A specific testing technique is applied after a construction step in a process in order to identify a particular type of defect. For example, load testing is a technique applied to determine how the product will respond to extreme volumes of work. It is applied to a system that has already been shown to compute correctly with a normal volume of work. The testing process specifies the range of testing activities to be used and when during the development process they will be applied.

The most important test artifacts that are produced during the testing process include

- test plans
- test cases and data sets
- test software and scripts
- test reports

This report describes techniques for producing these artifacts with sufficient quality to ensure their usefulness as product line assets. These test artifacts represent a sizable investment for most projects. Being able to include them in the asset base and apply them for all of the products in the product line represents an opportunity for substantial savings.

2.1.1 Test Plans

A test plan describes what is to be tested and how. Table 1 identifies the basic information that should be covered in a test plan as defined in the IEEE 829 standard. Column 2 in the table briefly describes the content of each section as defined in the standard. Column 3 describes how that content might be specialized to apply to product lines.

Table 1: Test Plan Sections

Test Plan Sections	Standard Section Content	Specialized Product Line Content
Introduction	Defines the complete test process for the artifacts within the defined scope	Nothing specific to product lines
Test Items	Defines the scope of the test plan	May be a specific asset, product, or the product line architecture
Tested Features	Specifies the features of the unit scheduled for completion during the current increment	• For a component, a subset of the component's interface • For a product, a specific set of use cases
Features Not Tested (per cycle)	Specifies the features not scheduled for completion during the current increment	• For a component, a subset of the component's interface • For a product, a specific set of use cases
Testing Strategy and Approach	Describes the test techniques and criteria used to ready the artifact for use in many contexts	See the techniques in Sections 3, 4, and 5.
Syntax	Specifies the syntax of the applicable notation so structural tests can be developed. Multiple notations are used in a project. The notations include representations for requirements, analysis, and design models as well as the code. Special notations such as component specification languages and architecture representations are also used.	There may be a special syntax/design notation to denote variation points or to relate a core asset to its instantiation in a specific product.

Table 1:   Test Plan Sections (continued)

Test Plan Sections	Standard Section Content	Specialized Product Line Content
Description of Functionality	Specifies pre/post-conditions for each behavior and invariants for each component	Defined for each component; may be made more specific for the instantiation of that component in a specific product
Arguments for Tests	Describes the levels and types of arguments used for assets within the scope of this plan	There will be two levels of arguments: • The first is the set of values for variation points to specify a particular product. • The second is the data values for a specific behavior.
Expected Output	Specifies the expected output at a level appropriate to the level of the test plan	For example, in an object-oriented system, outputs may be specified as abstract classes at the generic level and as specific objects at the concrete level. The generic product-test plan will have abstract specifications of the expected output. The test plan for a specific product will specialize each of the abstract specifications.
Specific Exclusions	Excludes certain inputs or certain functionality	A product-test plan will explicitly exclude those regions of commonality that have been tested at the product line level and that have no interaction with other components. Only those regions that have been modified will be retested.
Dependencies	Lists resources outside the scope that must still be available during a test	In a product line organization, there is a complex set of dependencies among versions of components, versions of products, and versions of test sets. The test plan must specify the specific configuration to be tested and the specific configuration of test assets. This may be a reference to a specific configuration defined in the configuration management (CM) tool.
Test-Case Success/ Failure Criteria	Lists criteria for how thoroughly to validate results such as checking for correctness of visible results and checking current state values at the end of a test	The description of the test case for the product line will define the expected result as specifically as possible. Some of the product line artifacts are complete as specified while others provide sufficient customizability to require a more abstract specification of expected results.

Table 1:   Test Plan Sections (continued)

Test Plan Sections	Standard Section Content	Specialized Product Line Content
Pass/Fail Criteria for the Complete Test Cycle	Defines "pass-forward," iterate, and "pass-back" criteria: sufficient test failures identifying certain types of defects lead to a pass backward to previous steps in the development process. Other types of failures cause iteration within the current development phase. Some failures may be sufficiently minor to allow a pass forward to the next phase.	Assets are reworked when a sufficient percentage of modifier methods create failures. The pass-backward action for a product reports failures found at the product level to the product line level so that the responsible asset can be modified.
Entrance Criteria/ Exit Criteria	Specifies criteria that must be met prior to starting testing or prior to handing the product off to an independent test organization. This is a link to the development process.	Each core asset must meet these criteria before being released to product teams.
Test Suspension Criteria and Resumption Requirements	See pass-back criteria above.	Nothing specific to product lines
Test Deliverables/ Status Communications Vehicles	Defines deliverables that include communication and feedback mechanisms. Each relationship is a bidirectional channel. Newly created assets are fed into a process and defects are fed backward.	The core-asset developers provide components to product developers. Defects found and associated with a specific component must be fed back to the asset developers.
Testing Tasks	Describes the planning, construction, and operation activities that must be accomplished to achieve the testing.	See Sections 3 and 4.
Hardware and Software Requirements	This must be comprehensive enough that artifacts are tested in all relevant contexts.	The product line architecture defines the external dependencies as well as the internal context for every component.
Problem Determination and Correction Responsibilities	Defines the interprocess relationships such as the relationship between the process steps for the construction and testing of components.	Product defects are handled by the product developers; core-asset defects are fed backward to the core-asset teams.
Staffing and Training Needs/ Assignments	See Section 2.4.	See Section 2.4.
Test Schedules	Defines a test schedule that specifies dependencies on the development schedule	Will be based on the artifact being created
Risks and Contingencies	Identifies the risks associated with different test strategies	The risks increase directly with the likelihood that this will be a core asset as opposed to a single-use artifact.
Approvals	Based on the scope of the plan	At either the product or product line level

In a product line organization, there are test plans for the individual components and for the individual products. These specific test plans are derived from templates. The product-test plans are derived from a template that is based on the product line architecture. The compo-

nent-test plans are derived from a template that is based on the component model used by the organization.

The component-test plan defines tests of the individual behaviors. The test plan for a larger component, or some other unit of integration, defines tests that exercise different combinations of those aggregated components' behaviors.

The relationships among developers and test plans are shown in Figure 2.

Core-asset developers	Component test plans and templates	Cluster test plans and templates	Test plans for system code and for non-code assets such as the architecture
	↓	↓	↓
Product developers	Specialized test plans	Integration of components	Product test plans
	Component	Subsystem	System
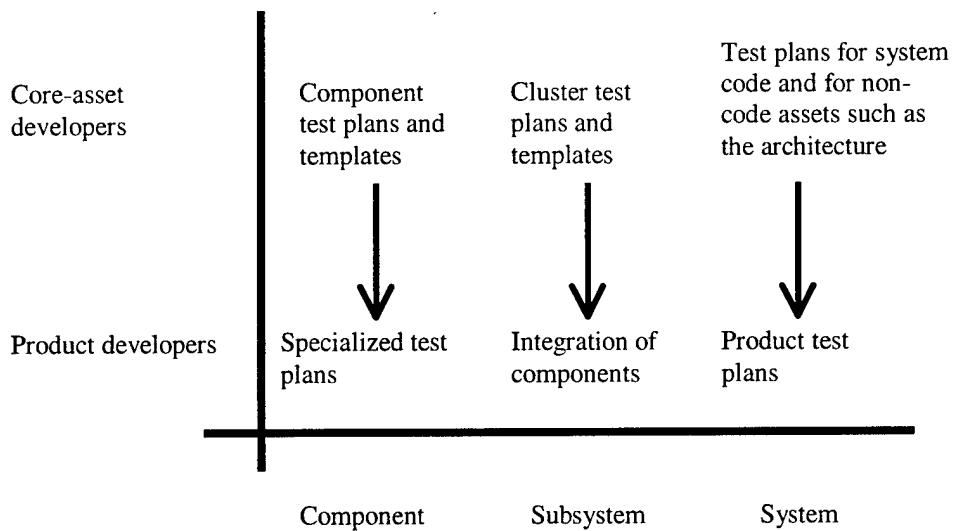
Figure 2: Relationships Among Developers and Test Plans

2.1.2 Test Cases

A test case has three parts: initial conditions, the input values, and the expected results. The initial conditions define the state of the artifact prior to the application of the test inputs. The input values provide a specific configuration of values that are associated with a specific set of behaviors given the specified initial state. The expected result portion of the test case describes how the artifact should respond to the stimulus given the specified initial state.

Test cases, such as those that require extensive databases, often require a lot of effort to prepare. A realistic database usually requires a large number of records. This makes it difficult to be certain of the expected results for individual test cases without manually evaluating all the records in the database. The product line environment provides opportunities to maximize the utility of these test cases, thus reducing the per-test cost. For example, the interface between a region of commonality and a set of variants provides for the reuse of interaction test information between the region of commonality and the individual variants.

In order to maximize the usefulness of the test cases across the product line. each test case must be associated with a specific component, or components, so that the inclusion of those assets in a product triggers the use of the associated test cases. This is accomplished by associating test cases with software specifications such as specific interfaces, specific design patterns or with product specifications such as use cases. Sections 3.3 and 4.1 discuss this in detail.

2.1.3 Test Software and Scripts

Test cases are used to "operate" the assets being tested. When the operation of the artifact is to be automated, the test cases are coded in some executable form. The scripting languages of test tools provide many of the same facilities as the programming languages used for development. The test scripts can be structured in a manner similar to the software of the system being developed. That is, test code can be parameterized based on the variation points defined in the product line architecture. Then product test cases can be assembled just as the product itself is assembled.

Test software may also be written in the same language as the product assets. The programmers constructing the product assets can be commissioned to produce the required test software. This usually includes harnesses that provide the execution environment for unit tests of individual components.

2.1.4 Test Reports

The results of applying test cases to the products under test are summarized in a test report. This report details the pass/fail status of each test case. The report summarizes the results in a way that developers can use to identify defects more quickly. For example, a variance analysis can be used to identify those components that have the highest probability of containing the defects that have led to failures. This analysis identifies the components that participated in the most failure scenarios.

Test reports are a different type of asset. They provide data for process improvement rather than a basis for building new products. Each report summarizes the types of defects found during the test activity that is the subject of the report. The source of these defects is determined through analysis. The gap between where the defect was found and where it was created is termed the defect live range. The smaller the gap, the more effective the test process is. By including this information in each test report, the effectiveness of the test processes used by the product line organization can be determined.

2.2 Qualifications for Testing Personnel

The test process and the personnel assigned to carry it out share a set of attributes that are different from those possessed by the personnel assigned to the development process [McGregor 00]. In particular, these personnel must be systematic, objective, thorough, and skeptical.

2.2.1 Systematic Personnel

The construction of test artifacts is an algorithmic process that follows repeatable procedures for selecting test cases and for executing them. This allows the people carrying out the testing activity to be certain about what they have and have not tested. Testing tasks are defined to identify the structure of the artifact and to traverse it in a traceable manner.

2.2.2 Objective Personnel

Testing techniques select test cases based only on the testing priorities established in the test plan. The construction of test artifacts is not biased by knowledge of the portions of the artifact in which the developer has the most confidence. Techniques such as employing test groups that are independent of the development team for higher-level tests and using buddy testing for developer-level testing achieve this objectivity.

2.2.3 Thorough Personnel

Testing techniques are designed to ensure that all aspects of an artifact are investigated. Sufficient test artifacts are created to achieve a previously specified degree of completeness in the coverage of the asset being tested. Completeness is a relative concept. Many different levels of coverage can be defined based on a variety of criteria to meet differing quality goals. For example, one level of coverage might be defined as

Test all variant values for each variant at least once during system testing.

A second, more thorough level of coverage might be stated as

Test all combinations of variant values across variants during system testing.

2.2.4 Skeptical Personnel

Testing artifacts are constructed with the minimum number of assumptions. Nothing is assumed to be correct until it has been proven to be correct through validation and verification. Products such as commercial off-the-shelf (COTS) components are subjected to an acceptance test before being incorporated into product line assets. The exceptions to this are those assets that have been certified by some accepted process and body.

2.3 Similar Structure Test Definition

A software product line organization must maintain a number of complex relationships. There are relationships between the group that develops the core assets and those who develop products, between the general product line architecture and specific product architectures, between versions of these architectures and versions of products, and many more as shown in Figure 3. As a general rule, structuring the relationships among test artifacts to mirror the structure of the relationships among production artifacts results in a more efficient implementation and minimizes maintenance.

2.3.1 Down the Product/Product Line Relationship

A major relationship in a product line organization is that between the general product line and specific products. Each product specializes the general product line architecture to solve a specific problem. The nature of this relationship is shaped largely by the choices made at the individual variation points. Figure 3 illustrates this relationship between the product line architecture and a specific product (Product A).
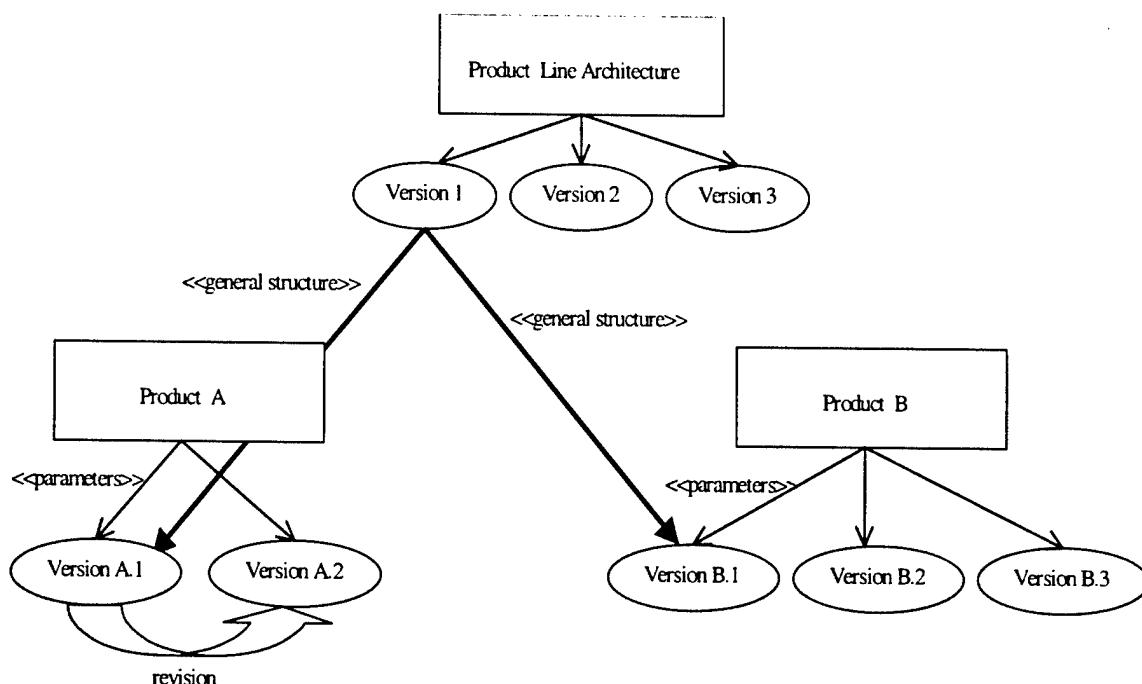
Figure 3: *Product Line Relationships*

Test cases are created at the product line level and then specialized for each product. This specialization may affect each part of the test case or only select portions. Since the goal of specialization in the products is to maintain the behavior compatibility of products to a product line, the same will hold true for the test cases. This relationship is illustrated in Figure 4.
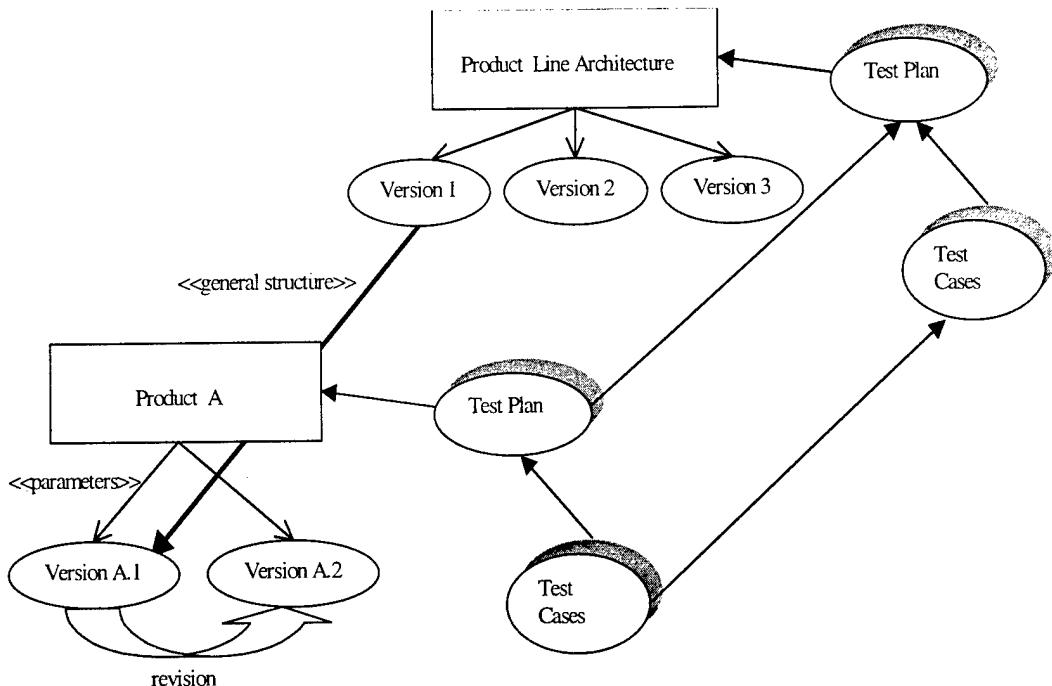
Figure 4: *Relationships with Testing Artifacts*

The test case for this relationship has these three parts:

- initial conditions - The initial conditions can become more complete. A test case in the wireless example might have an initial condition, at the product line level, that states

  The user has selected a phone number from the address book to dial.

  In the product-specific test case this becomes

  The user has used voice commands to select a phone number from the address book.

- inputs - The number of inputs in a test case may be reduced in a product because certain variants have been selected.

- expected results - The expected results contained in a product line's generic test case may have to be edited to make them more explicit. At the product line level, the expected result might be

  The phone call has been dialed and any required information has been updated.

  At the product level this becomes

  The phone call has been dialed and the timestamp for this attempt is entered into the address book.

2.3.2 Across Versions and Products

Test cases must be used across versions of a product and across multiple products, as shown in Figure 5. As modifications are made to software to create a new version, complementary changes must be made to the test cases. Section 5.1.1 discusses establishing traceability between a specification and its test cases.
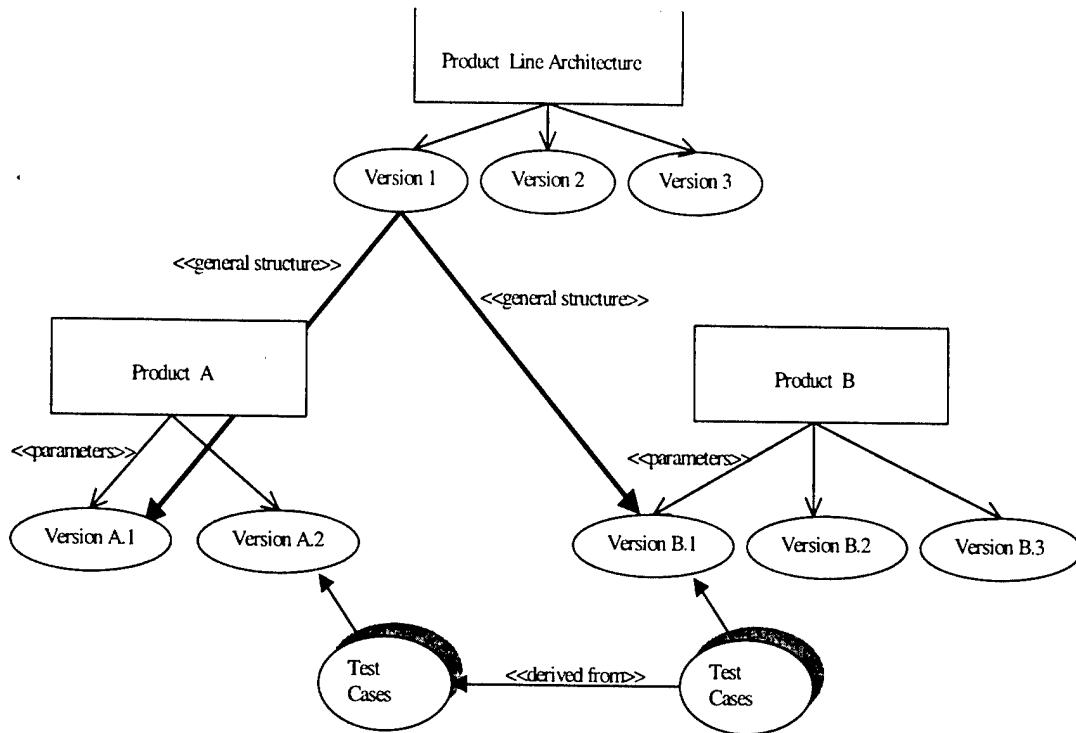
Figure 5: Across Products Relationships

Test cases can be used across products by maintaining a product line repository and promoting product-level test cases to the product line level. This is particularly useful at the unit level where the complete package for a component includes its test suite.

At the product level for Product A, one test case that was developed was

    The user creates a new appointment and saves it.

Next, it is generalized up to the product line level as

    The user inputs data to an application and stores it.

This is then specialized in Product B, as the test case:

    The user adds a new entry to the address book and saves it.

2.4 Roles and Responsibilities

The testing and development processes are closely related with information flowing between roles and responsibilities related to testing activities. Roles in both processes, which are related to testing, are described here.

Often personnel are assigned multiple roles and the accompanying responsibilities. The size of the organization and complexity of the products help to determine the scope of each person's role.

2.4.1 Testing Roles

Each of these roles is present at both the product line and product levels of the organization. Since this is really just a matter of difference in scope and priority rather than difference in tasks, each role is defined only once.

2.4.1.1 Test Architect

The test software for a product line is sufficiently extensive and complex to justify the role of a test architect. The test architect role parallels that of the product software architect, carrying the responsibility for achieving the qualities defined in the test plan. The architect defines the structure of the test software and defines the basic test assets. This test software architecture can then be specialized to apply to the specific environment, qualities, and constraints of a specific product.

Test Architect in the Wireless Example

The product line of wireless devices would require testing software that operates in a hardware simulation environment and eventually migrates to the test harness on the integrated hardware and software platform. The test software must be easily ported to different processors as new models of wireless devices migrate to the latest and greatest processor.

2.4.1.2 Tester

The tester defines, implements, and executes specific test cases and appears at each point in the development process where testing is performed. In many organizations these responsibilities may be divided among two or more people. For example, the core-asset team may define a set of product test cases based on the product line requirements model, but a product builder will actually execute the tests.

The skills needed by a tester vary from one organization to another and even from one type of testing to another within the same organization. The tester may have the same programming skills as a developer and use the same programming language as is being used in the produc-

tion software, or the tester may use the simpler scripting language of a test tool that requires less programming skills. The tester may specify test conditions and then commission software from the development staff.

Tester in the Wireless Example

A tester in the wireless product line would define test cases that exercise the interface between the software and hardware. Tests are defined for the interface between the user and the device. The test cases exploring the phone-user interface can be more explicitly defined at the product line level than those for the hardware-software interface. The phone-user test cases would have to be supplemented with application-specific cases. The hardware-software test cases would require more extensive modification when functionality migrates from software into the hardware.

2.4.1.3 Test Manager

The test manager is responsible for providing the resources necessary to achieve the qualities required by the test plan. The test manager analyzes the required qualities and associated mechanisms and estimates the resources needed.

Test Manager in the Wireless Example

The test plan for the wireless device product line calls for sufficient test cases to achieve an n-way switch cover [Chow 78] for the protocol engine in the device. The test manager, using data from previous projects and the state machine for the protocol engine, would estimate the number of test cases required for achieving the required coverage.

2.4.2 Development Roles

The most significant information flow from development to testing involves specification information. Although code also flows from development to testing, there is little interaction revolving around this information.

2.4.2.1 Specifier

The specifier is responsible for defining what an artifact is intended to do and must produce a complete and correct specification that is consistent with related specifications.

The most likely error committed in this role is confusion between requirement and design information. For example, the specifier may require that a wireless device

shall provide a button on the screen by which a call may be terminated

instead of

shall provide a visual prompt by which the caller may terminate a call

During inspection, each constraint should be examined to determine whether it describes a desired feature or defines how to achieve that feature.

2.5 Combinatorial Test Design

A product line design has numerous points at which the architecture can be varied to produce different products. These variations begin as requirements for different but related product features and lead to architectural points of variation. These points of variation are propagated down to the method level where the number and types of parameters are varied. It is virtually never possible to test all the combinations of these variations. At these various points, choices must be made about which values to use during a specific test. This section describes the application of a class of techniques to this problem.

Combinatorial test designs support the construction of test cases by identifying various levels of combinations of input values for the asset under test. This approach is based on a simple process:

1. Identify attributes that should vary from one test to another. At the component-test level, each parameter to a method is such an attribute. At the interface level, related methods may have different signatures based on varying the number and type of parameters. At the system level it is an architectural variation point. These attributes will be termed factors.

2. For each factor, identify the set of possible values that the factor may have. This can be based on an equivalence-class analysis of the data type and context of each parameter. A common set of equivalence classes used for the Integer data type is negatives, positives, and zero. Variation points are treated like discrete types; each variation is an equivalence class by itself. The factor has as its value one of the discrete values, such as a specific signature for a selected method, at each point in the test execution. These values of the factor will be termed its levels.

3. Apply a combinatorial design algorithm to select levels for each of the factors in various combinations. That is, given two points of variation in an architecture, one with two existing variants and the other with three, a test design algorithm would select a variant of the first variation point and a variant of the second variation point. An exhaustive test set would cover all possible combinations of variants.

The number of variation points and the number of different variants at each point make an exhaustive test set much too large to be practical. Suppose there are 13 variation points (factors), each of which has three possible variants (levels). There are 1,594,323 possible combinations of variants. Combinatorial design allows the systematic selection of a less-than-exhaustive test set. By selecting all pair-wise combinations, as opposed to all possible combinations, the number of test cases is dramatically reduced. Cohen and associates [Cohen 96] selected 15 tests that cover all possible pair-wise combinations of the 13 factors.

Phadke's Design of Experiments technique [Phadke 89] uses orthogonal arrays as a basis for designing test sets. Research has shown that while achieving tremendous compressions of the number of required test cases, the reduced test set retains 90% of the effectiveness of an exhaustive test set.

Consider the options for the product line of wireless devices, described briefly in Appendix A. The product line produces software to run on three different handsets, identified here as A, B, and C to avoid brand names. Each device has a specified coverage area: local, national, and international. Each device receives one of three possible protocol sets: GPRS, EGPRS, or UMTS. An exhaustive test set for these three factors, each with three levels, would require 27 test cases.

Table 2 shows an orthogonal array that addresses pair-wise combinations and uses nine test cases to cover three factors with three levels.

Table 2:   Orthogonal Array

Factor$_1$	Factor$_2$	Factor$_3$
1	1	3
2	1	2
3	1	1
1	2	2
2	2	1
3	2	3
1	3	1
2	3	3
3	3	2

Each level of each factor is assigned a number between one and three. Then those level values are substituted for the corresponding number in the original array. For the example, the mapping between the problem and the orthogonal array can be given as

```
{{(1,A)(2,B)(3,C)}{(1,GPRS)(2,EGPRS)(3,UMTS)}{(1,local)(2,national)
(3,international)}}
```

The resulting array has the level values mapped onto the values in the orthogonal array. For the wireless device example, the resulting array is shown in Table 3.

Table 3:   Mapped Array

Handset Manufacturer	Protocol Set	Coverage Area
A	GPRS	international
B	GPRS	national
C	GPRS	local
A	EGPRS	national
B	EGPRS	local
C	EGPRS	international
A	UMTS	local
B	UMTS	international
C	UMTS	national

2.6 Testability

Testability is the degree to which an artifact will reveal its defects when tested. The more testable a piece of software is, the smaller the effort required to locate defects will be. Testability in software is provided by making the internal state of the system *observable* and by making the software under test *controllable*. Test cases must be able to control the software, place it in specific states, and then observe the effects on the internal state of the system.

Testability in a product line effort can be achieved in a number of ways. The product line architecture will define certain interfaces that provide access to the state of an executing system. Other test interfaces may be defined that every component or at least any high-level component must implement. Design guidelines may require accessor methods for all state attributes of a component. Other techniques such as providing test harnesses that use techniques such as meta-level programming methods give complete observability and greatly improve controllability.

2.7 Coverage

A test coverage metric measures how thoroughly an artifact has been tested. Consider a method that takes a single Boolean value as input and uses no other data in its computation. Running two test cases, one with an input of *true* and another with input of *false*, has exhausted all possible inputs in the specification. This test set has achieved 100% functional test coverage of the method. This level of coverage assures that the asset correctly addresses the complete range of its specification.

Functional coverage does not answer any questions about how well the actual implementation of the asset has been covered. An analysis of which lines of code were executed by the test cases or which decision paths were taken by the tests is needed to determine the degree of structural test coverage that has been achieved. Achieving 100% structural coverage assures that the asset does nothing it is not supposed to do, but does not assure that it does *all* that it is supposed to do.

The cost of testing an artifact in a specific domain depends upon the level of test coverage that the test plan defines as adequate. If the test plan for the above method had set 50% as the desired level of coverage, a single test case would have been sufficient and the cost of testing would be cut in half.

What is considered "adequate testing" will vary from one organization to another. Industries that develop life-critical systems will specify higher levels of coverage than those that will develop mission-critical systems that do not affect human life. Industries that are regulated, such as medical equipment or aircraft control, will specify more complete coverage than industries that are unregulated.

A product line organization will establish coverage levels for each type of artifact in the test plan for that artifact. The criticality of the product line's domain (i.e., life, safety or mission critical) will determine the basic approach to specifying coverage. Components produced by the product line team to be used in a variety of products will be tested to higher levels of coverage than those produced by product teams to serve a specific need in a single product. This is necessary since the deployment environment of a product line asset is not specified as narrowly as the deployment environment for a specific product is.

Product line organizations can achieve very high levels of coverage by aggregating the test executions performed by the individual product-test teams. The test team for the product line will have conducted a unit test on a component that achieves some level of coverage of the underlying implementation. It may well not achieve 100% structural coverage. As the product teams integrate the component into their products and conduct product tests, some lines of the component will be executed. By tracking the lines executed by all product teams, a very high percentage of the code will be covered.

2.8 Summary

The relationship between the team that develops core assets and the individual product teams provides a context within which to structure the definitions of the basic testing artifacts. The content of each testing artifact is the same for product line organizations as it is for any software development effort. However, there are opportunities for economies of scale and economies of scope that reduce significantly the cost per test case if the dependencies between the production products and testing artifacts are correctly managed. This is managed by making the pieces more modular so that a product-test team can compose its artifacts from core assets.

The descriptions of specific types of tests described in the next sections will provide specific guidelines for many of the characteristics discussed in this section. A number of roles are required in the testing effort. The personnel assigned to these roles will depend upon the phase in the development process where the tests are conducted. The levels of coverage will also vary from one development phase to another. In fact, coverage should become more demanding with successive iterations over the same phase.

3 Testing Assets

The number of variation points and possible values for each variation make the testing of all possible products that can be built from the product line impossible. This makes it imperative that products be composed of high-quality components [Ardis 00].

The term "asset" covers a very wide range of artifacts, not all of which are executable pieces of code. Architectures, designs, plans, and documentation are core assets. This section describes techniques for testing individual assets whether or not they are executable.

The core assets are created and tested by the core-asset teams. Specialized versions of these assets are created and tested for individual products. These new assets are usually tested by the product team that creates them. The artifacts created to support testing assets are themselves assets and are incorporated into the core-asset base of the product line so that other teams may benefit from them. In this section the focus is on testing an asset prior to its integration into large units regardless of the context.

3.1 Asset Specification

A product line organization communicates between teams and between development phases using specifications as the medium of communication. A specification is a description of what something is supposed to be without constraining how the specification will be realized. The product line architecture is a blend of specification and design information for a completed product. The content of the architecture includes specifications for the components that comprise the product.

One important aspect of testing is the comparison of an artifact with its specification. The testing process includes activities that verify that the artifact's specification conforms to its requirements and that an implementation correctly implements the specification. In the absence of a specification, the tester is forced to derive a description from the context and content of the artifact. It takes longer for the tester to do this than it would have taken the artifact producer, and the resulting description is almost certainly a less accurate view of what the asset is supposed to be.

3.1.1 Syntactic Specification

Certain artifacts, such as the test plan template shown in Figure 2, are specified structurally in the product development process. These specifications are limited to a syntactic level of detail. The test plan for a component or product is based on the format specified in the test process. The information that goes into the plan for this type of specification is based on the structure of the document rather than the semantics of the system.

When artifacts realize this type of specification, they are usually inspected for conformance to the standard document structure. Are the required fields present? Do they contain information? The information is verified in an ad hoc manner, if at all. The completeness and correctness of that verification relies on the expertise of the specific inspector.

Inspection for conformance to syntactic specification is useful for a couple of purposes. First, it provides an audit to determine whether policies and processes are being followed. For example, the "form follows form" policy can be checked to determine whether there is a corresponding test set for each artifact. Second, this type of inspection allows new personnel to become familiar with the work products they are expected to produce.

3.1.2 Semantic Specification

Artifacts that represent the semantics of the products and core assets, such as the product line architecture and components, are the realizations of specifications that are created as part of the development process. For example, the developers may document the specification of components using a notation such as the Unified Modeling Language (UML) [Rumbaugh 99]. The notation needs to be sufficiently comprehensive to represent all of the specification needs of the product line organization. The notation must be able to express the static relationships between concepts and, at the same time, describe the dynamic constraints on run-time behavior.

An "interface" is a language vehicle for associating part of a specification with specific code modules. An interface typically contains the specification of a portion of those behaviors that are available to other modules. The interface mechanism provides the means to separate the specification information from the implementation of those behaviors and to partition the total specification into logical groups of behaviors. Each grouping contains semantically related behaviors. The interface mechanism will be used as an organizing principle for reusing test information associated with the individual behaviors specified in that interface.

Techniques such as Design by Contract [Meyer 97] offer a means to specify the individual behaviors named in an interface. Design by Contract defines pre-conditions that specify when a behavior may be used, post-conditions that specify a guarantee of service, and invariants that specify the static operating limits of the unit containing the behavior. The Object Con-

straint Language (OCL), a component of UML, can be used to state these specifications [Warmer 99]. Because these are logical constraints, the relationships between software units (e.g., inheritance) can be used to systematically derive constraints with variants of the original specification.

3.1.3 Specification of Qualities

Part of the specification of the product line architecture is a prioritized list of qualities. These specifications do not define functionality for a product. Nevertheless the presence of these qualities in the completed products is necessary for the product line's success. While there are no widely accepted notations for this type of specification, the attribute-based architecture styles reported by Clements and Northrop [Clements 02a] do include techniques for verifying the presence of specific properties.

3.1.4 Testing and Specifications

Functional test cases are created from the information in a semantic specification. Therefore, these specifications must be correct and complete. During architecture testing, described in Section 3.2.2, the specifications of the high-level components will be examined. The development process should also define test points where the specifications of lower-level components, which are composed to produce the high-level components, are inspected. If the specification of a component is not correct, the derived test cases will define inaccurate expected results. If the specification is not complete, the component developer with domain knowledge may add obviously needed behaviors to the implementation without adding them to the specification. The test cases derived from this specification will not be able to verify that every behavior of the component is correct.

The specifications contained in the product line architecture will be the basis for numerous implementations that realize the specification in somewhat different ways. Structural test cases are created from the structure of each of these implementations. These tests provide a means to determine whether the implementation does anything that it is not supposed to do. The relationship between a specification and its implementations is critical to the success of the components that aggregate those implementations.

3.1.5 Costs of Not Creating Specifications

In a product line organization, where components are used many times in many environments, a complete, correct, and consistent specification is essential. Specification is a means of determining what an artifact is supposed to do and then communicating that information to others. It takes some time to determine what services the asset should provide and then some additional time to create the written specification. Not using a written specification only eliminates the time required to represent the information, and it actually increases the time

needed by development personnel to determine what the asset should do. Added to this is the time required for testers to infer what the developer intended.

3.2 Static Testing

Many product line assets are not executable code. These assets must be verified against the appropriate specification but without the benefit of executing code. Documents such as plans will be verified against a syntactic specification that defines the structure of the artifact. The products of analysis and design are verified against the semantic specification.

3.2.1 Inspections

Software inspections are strict and close examinations conducted on specifications, design, code, test, and other artifacts [Ebenau 94]. Where design reviews are intended to review progress and obtain feedback [Pfleeger 98, page 242], inspections are intended to identify defects. Inspections are highly effective at this task. Teams with initial training will find an average of 50% of the defects present, and those with experience will find 60% to 90% of them [O'Neill 89].

Fagan [Fagan 76] introduced inspections to IBM. His approach included a set of checklists that are specific to the type of artifact being inspected. McGregor [McGregor 01] added the concept of guiding the inspection process using high-level test cases selected based on the semantics of the system being developed. This approach allows test coverage measures to determine when the inspection is sufficient. Both of these techniques add to the cost of the static-testing process but also contribute to its effectiveness.

3.2.1.1 Inspection Criteria

In general, inspections are intended to determine whether an artifact is correct and complete. A product line organization inspects an artifact to determine whether it is consistent with other artifacts. The product line organization constructs a product by combining a number of choices at specific variation points. This results in a higher probability that there are contradictions among artifacts. The criteria can be defined more specifically as follows:

- An artifact is correct if it matches some standard deemed accurate by experts.
- An artifact is complete if it addresses the full range of possible values as defined in its specification or the product line scope document.
- An artifact is consistent if it does not contain any contradictions among its internal components and does not contradict any other product line artifact.

Inspections may have additional criteria such as the architectural qualities that are investigated during the inspection process.

3.2.1.2 Inspection Costs

Inspections are a human-intensive activity whose benefits must be considered in terms of the number of person-hours they require. The costs for an inspection program include

- materials preparation
- training
- conduct of the inspection
- report preparation

After the cost of conducting the inspection, the cost of creating materials is the most expensive part of the process. The materials needed for a Fagan inspection [Fagan 76] are checklists that define specific criteria for specific artifacts. For example, a design checklist is used by the inspector to remember the rules of design practice against which to check the artifact. These checklists must be modified for different types of products, for example, a real-time system versus an e-business system. The close relationship among products in a product line minimizes the need for checklist modifications.

The cost of the Fagan inspection is more than compensated for by the reduced costs of defect repair. Companies have reported savings of tenfold between repairing a requirement defect found at system test time versus the same defect found during the high-level design inspection.

The Guided Inspection process that McGregor describes [McGregor 01] adds the cost of preparing high-level test cases to the cost of the Fagan inspection. These test cases are based on use cases. Use cases are first defined at the product line level and then specialized to products. The test cases are likewise first developed as high-level scenarios and then expanded into more specific scenarios that are applicable to a specific product.

A Guided Inspection provides the same cost savings benefits as Fagan inspections but with an improved power to "guide" the inspection process. The test cases are selected to emphasize specific types of defects or to verify high-priority specifications. This approach also reduces the cost of preparing the executable test cases that eventually are used to validate the code produced by core-asset builders and product builders. The high-level scenarios that are used for the inspection are made more specific and result in the system-level test cases for each product.

A software product line organization has the opportunity to amortize the cost of inspection preparation across a number of products. By inspecting product line artifacts, the benefits of the inspection activity can be shared across all of the products that use the resulting asset. This further magnifies the savings of detecting defects early that has already been reported by numerous studies [O'Neill 89] and more than offsets the costs of an inspection program.

3.2.2 Architecture Evaluation

Testing the product line's architecture has the potential to impact the quality of all of the products developed in the product line. The inspection of the architecture is sufficiently important to be treated in its own section in this report and as a practice area in the conceptual framework developed by the SEI [Clements 02b]. The architecture is evaluated using the three criteria discussed in the previous section. Additionally, the architecture's ability to achieve each quality attribute is validated. This requirement is satisfied through one or more testing activities applied at specific points in the process. For example, if performance is an important quality to a project, specific performance-testing activities are applied during the architecture inspection, system integration, and customer acceptance phases of product development.

The Architecture Tradeoff Analysis Method[SM] [Clements 02a] and Guided Inspection [McGregor 01] both use scenarios as the basis for the analysis of the architecture. These scenarios are constructed to address the requirements of the products for which this will be the architecture and the specific qualities important to a specific architecture. The scenarios are used to analyze the decisions made in bringing the architecture to its current state. The analysis determines the effect of those decisions on the attributes that are the focus of the scenarios. In short, these scenarios play the role of test cases. The expected result is that the architecture exhibits the qualities for which it is being tested. This may be exhibited through a detailed analysis of a non-executable model or through the execution of an executable prototype.

This testing perspective on architecture evaluation allows the evaluation to be guided by coverage criteria that determine when sufficient scenarios have been analyzed. A minimal evaluation will derive one scenario for each requirement; this may translate into one per use case and one for each system quality. A more thorough evaluation will use multiple scenarios from each requirement and include failure cases as well as success cases. The number of scenarios derived for each requirement should be based on the priority of that requirement as documented in the use case.

The number of test scenarios that should be created for "adequate" testing is related to the criticality of the domain, such as whether a requirement is life or mission critical, and the complexity of the architecture. The complexity is reflected in the number of interconnections between components in the architecture. Selecting a sufficient number of scenarios to navigate each interconnection is a basic level of the architecture's structural coverage.

All product line architectures must have the quality of being sufficiently flexible to support all possible combinations of choices at all variation points. This flexibility is constrained by

---

[SM] Architecture Tradeoff Analysis Method and ATAM are service mark of Carnegie Mellon University.

the scope of the product line that limits the possible variations. The type of scenario that is selected for flexibility testing is one in which the least likely variant is selected at each variation point and all optional features are included. This "extreme" product is analyzed to determine whether the architecture can realize this particular configuration.

3.2.3 Summary

Inspection is used at several points in the product line development process. Table 4 briefly describes the test points in a typical development process, summarizes the test techniques, and defines multiple levels of coverage. Increased coverage results in increased detection of defects.

Table 4: Static-Testing Techniques

Asset	Test Technique	Coverage Measure	
Requirements model	Inspection by a team of domain experts who have not participated in developing the requirements The team develops a set of scenarios that define its visions for the system.	1. Every use case should be touched by at least one of the expert's scenarios. 2. Each variation point is sampled with multiple scenarios.	
Analysis model	Inspection by a team of domain experts who created the requirements and designers who will use the analysis model as input to architectural design	1. One test scenario for each use case's default "usual course" 2. One test scenario for several highly probable variants of the use case's "usual course" 3. Test set expanded to include test scenarios for the use case's alternative and exceptional course	
Architecture	Inspection by a team of analysts who created the analysis model and designers who will use the architecture model as input to detailed design An executable model may be used instead of a manual inspection if it is available.	1. One test scenario for each use case's default "usual course" 2. One test scenario for several highly probable variants of the use case's "usual course" 3. Test set expanded to include test scenarios for the use case's alternative and exceptional course 4. One test scenario for each identified architectural quality	
Detailed design	Inspection by a team of architects who created the architecture model and developers who will code the interface implementations The quality scenarios are used to guide a more in-depth analysis of the design. A syntax checker can be used if the Object Constraint Language or other parsable specification language is used.	1. One test scenario for each use case's default "usual course" 2. One test scenario for several highly probable variants of the use case's "usual course" 3. Test scenarios for the use case's alternative and exceptional course 4. Test scenarios for architectural qualities are re-analyzed.	

3.3 Dynamic Testing

The software artifacts produced by the product line organization can be executed against specific test cases. These tests are applied at several points as the code is constructed. Beizer lists three major divisions in dynamic testing: unit, integration, and system [Beizer 90], each of which is described below.

3.3.1 Unit Testing

Unit testing is defined as

> "*the testing of individual hardware or software units or groups of related units*"
> *[IEEE 90].*

The focus of unit testing is on the fundamental units of development whether this is a procedure or an object. The purpose of this level of testing is to determine whether the unit correctly and completely implements its specification. This is accomplished by functional tests created from the specification. Often it is also desirable to determine that the software does not do anything that it should not do. This can be accomplished using structural tests that are based on the structure of the unit's implementation.

As described in Section 3.1, the specification of a unit is typically comprised of a set of interface definitions. Each interface defines a set of behaviors and a set of quality attributes that the unit must satisfy. An atomic unit, one that is defined in terms of language primitives, is included only in products for which its specific behaviors are required. The interface of a unit, which encapsulates one or more variation points, will specify parameters that determine which variant will be used. The level of variation within the unit specification is reflected in the specification of the test cases. This will be discussed in detail in Section 5.

The unit's interface contains specifications for quality attributes. For example, there may be performance requirements on the storage and retrieval behaviors of a container component. Test cases for these qualities often require the establishment of complex test conditions for which the product line test organization will provide test drivers. For example, a component that has a security requirement would be tested using a harness that can generate security credentials and pass them to the unit under test as part of each test case.

Those who develop the core assets assume most of the responsibility for unit testing. The unit-test suite will provide evidence of conformance to the specification, as well as correctness and reliability, to the product teams. Product developers will examine the test plans to determine whether the test coverage provided by the plan is sufficient for their needs or whether additional tests should be run.

Unit-test plans, test cases, and reports become part of the documentation that accompanies a core asset. The unit-test plan is inspected for conformance to the test plan format and for adequate coverage based on the product line test plan. The test cases are inspected to assure that they achieve the specified level of coverage and that they correctly state the expected results. The test reports are inspected for conformance to the test plan's format.

The unit tests are reused across versions of the product line component set and down into specific products. The software that implements the tests is structured so that relationships between units in the production software are mirrored in the test software. This supports traceability between test and production software units.

In the wireless device product line, a set of components represents the individual applications that run on the device. For example, a set of productivity tools is implemented as a set of components. Each component provides some unique behavior but all must implement a fundamental interface specification that allows the sharing of information among all of the applications (e.g., clipboard actions). The unit-test suite for one of these components would aggregate into it the functional test set for the `productivityTool` interface. This test set would be a reusable asset that could be used by each developer of a component that implements that interface.

The specification of one of the methods in the `productivityTool` interface is shown in simplified form as

```
Pre-condition: clipboard.hasContent()
Text      copyFromClipboard();
Post-condition: text::Text = clipboard.getContent()
```

Functional tests can be developed from this specification and then applied to any component that implements the `productivityTool` interface.

3.3.2 Integration Testing

Integration testing is defined as

> "testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them" [IEEE 90].

The focus of integration testing is on those interactions that occur between tested units. One unit may request a service from another unit that it has not implemented; the unit may provide an incorrect parameter in a message to another unit, or the thread of control in one unit may not be synchronized with a thread in another unit.

Integration testing is a cumulative effort. As development proceeds, the size of the pieces to be integrated steadily increases. As long as testing begins with the smallest pieces, each new round of integration has a well-defined set of interactions that must be tested and a stable base upon which to build.

Integration is more than just a development-process phase that integrates tested units. Integration is a shared responsibility between the core-assets builders and the product builders. Units are built by integrating other units. This type of integration is tested via the unit test of the integrating unit. At some point, a unit is useful in building a specific product. At that point the product team takes over responsibility and tests the integration of that unit into the product. This integration and integration testing continues iteratively until the integrated units form a completed product.

Product line organizations may produce a number of variants for use at each variation point. This makes it impossible to test the integration of all possible combinations of all variations. Two techniques can be used to mitigate this problem:

- Combinatorial test designs, discussed in Section 2.5, can be used to greatly reduce the number of combinations that need to be tested.
- If integration testing is performed incrementally, the number of combinations that must be covered is much smaller.

One factor that determines the number of tests that must be executed is the number and length of protocol sequences. A protocol is an interaction between two or more units with messages going back and forth between units. A scenario in which messages are passed in both directions between the two pieces being integrated represents a protocol. These scenarios must be created from an analysis of the pieces being integrated since the scenarios are not end-user behaviors.

For example, in a wireless communication device, one component controls the display. It does this in response to signals from several components that produce information for the user. The display component also sends signals to these producer components based on touches to the screen. One protocol would be the display of a menu, the selection of an entry by the user, the notification of the producer about the selection, and then the display of a submenu, as shown in Figure 6.
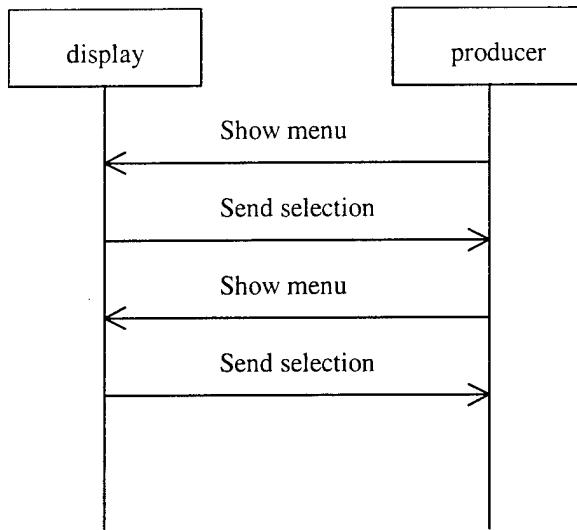
Figure 6:   *Protocol for Menu Hierarchy Traversal*

3.3.3 System Testing

System testing is defined as

> *"testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements"* *[IEEE 90]*

The focus of system-level testing shifts away from basic correctness toward conformance to requirements for a specific product. This level of testing is discussed in more detail in Section 4. Dynamic-testing techniques are described in Table 5.

Table 5:   Dynamic-Testing Techniques

Asset	Test Technique	Example Coverage Measures
Component	Functional tests - based on the specification	1. One test case per method 2. One test case per post-condition clause
Component	Structural tests - based on the structure of the implementation	1. Every statement executed 2. Every branch exit executed
Component	Interaction tests - based on two pieces of code "touching" the same variable	1. One test case per interaction 2. Test cases that reverse interactions to check for sequencing errors
Cluster	Protocol testing of a series of messages	1. One test case per protocol 2. One test case for each protocol sequence
Product	Functional tests - based on requirements	1. One test case for each combination of equivalence class values from a use-case usual course 2. One test case for each combination of equivalence class values from each use-case usual, alternative, and exceptional course

3.3.4 Costs of Dynamic Testing

Traditionally, the expenditures for dynamic testing have been heavily weighted toward the system test activities with the unit-test activities receiving very little attention and integration testing receiving only a little more. More recently the benefits of testing as early as possible have become obvious, and resources have been shifted to the earliest static testing of requirements and architecture models. This reduces the need for, and hence the cost of, dynamic testing.

Product line organizations can amortize the costs of the automation software, such as specialized harnesses and simulators, which is needed for dynamic testing. In cases where domain-specific languages are created to specify and construct the products, companion test tools should be created.

The costs of dynamic testing can be traded off against the correctness of the application. Products within the same product line can be developed to differing levels of correctness, and for differing levels of cost, by varying the test coverage specified in the product's test plan.

3.4 Regression-Test Strategies

Regression testing is defined as

"selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements" [IEEE 90].

Software product line organizations most often use an iterative, incremental development approach that schedules the rework of assets several times during the development process. Each time a piece of software is revisited, it is possible that new defects will become visible in those parts of the asset that are not modified. A portion of the test suite that is applied after the rework is intended to test those parts of the software that were not modified during the rework. The purpose of these test cases is to assure that previously working software has not regressed into a non-working state.

When product teams use core assets to create products, there is often a need to modify an asset to achieve an exact fit. These modified assets are tested using a blend of regression and development testing. The modified portions of the asset should be exercised using

- existing functional tests if the specification of the asset has not changed

- newly created functional tests if the modifications are the result of a changed specification

- structural tests that are created to reflect the new code

The unmodified portion of the asset is tested using a regression-test set that is chosen from the existing tests for that asset. The regression set should be chosen to test any method that calls a modified method or any method that uses a data attribute that has been modified or that is also used by the modified code.

Regression testing is a technique rather than a phase in the development process. It is the approach used to guide a testing activity after the initial performance of that activity. There are two important decisions that characterize regression testing:

- decisions about which existing tests will be used in the regression test

- decisions about how this testing is automated.

3.4.1 Selection of Regression-Test Cases

Since regression testing assumes that the artifact has been tested previously, there is an existing test suite. If the test application is totally automated and cheap to use, the regression-test set may simply be the entire original test set. Because this is seldom possible, regression-test selection is a strategy for sampling from existing test sets to build smaller test sets.

The sampling method should retain as much of the defect detection power of the original test suite as possible. The regression-test suite should also be structured to focus on the high-priority, high-level test cases. The selection of test cases for an asset should be based on the use of the artifact being tested and on the part of the artifact that has been modified since the last test activity.

- Include those tests that failed in the previous round of testing. This is particularly an issue if the team uses an approach in which the test harness automatically generates the specific input values for a test case from a template every time the test is applied. The regression strategy must be able to guarantee that at least the test cases that caused failures are reapplied to verify that the defect causing the failure has been repaired. This requires storing test cases that cause failures and reapplying those specific cases.

- Include tests that cover the modifications made to the code. If the modifications are a bug fix, select the tests from the structural test suite to cover specific lines of code. This may involve "diffing" the current version with the previous one or reading developer comments at the head of affected files. If the modification was a change in specification, select functional test cases that cover that facet of the asset. Report all bugs and bug fixes to the core-asset team.

- Include tests of the unmodified portion of the asset such as methods that call modified methods and methods that use data attributes that are also used by modified methods.

3.4.2 Test Automation

The application of test cases to the artifacts under test in the regression-test suite should be automated as much as possible. Section 5.2 provides a discussion of automating all types of testing.

3.4.3 Inventory Maintenance

Periodically the entire software asset base should be regression tested. This is the software equivalent of "freshening" inventory. The core-asset development team maintains a test suite that has been selected from the functional suites that were developed for each interface specification. In particular, as tests are created to test specific modifications, they are added to the inventory test suite. This round of testing identifies those components that have not been overlooked for bug fixes and specification modifications.

3.5 Operational Profile for an Asset

An operational profile is a description of how frequently each behavior is executed in a specific environment. A profile is a list of probabilities based on the relative frequency of use. The probabilities are used to determine how many test cases will be created and executed for each particular feature.

For an asset such as a component, the specifier or tester determines the relative frequency of use for each method in an interface. The tester then writes different numbers of test cases per behavior based on the profile. Consider the following example component specification:

```
Component BlueToothDeviceDriver{
        public send(DataBuffer db);
        public receive(DataBuffer db);
        public test();
}
```

The `test` behavior will be used much less than the `send` and `receive` behaviors that are the main actions of the component. How much less is difficult to quantify exactly. The tester may choose to estimate such as: `test` will only be used 10% of the time with `send` and `receive` being used equally often. This leads to the operational profile: ((send, 0.45), (receive, 0.45), (test, 0.1)). One hundred test cases would be divided as 45 test cases each for `send` and `receive` and 10 test cases for `test`.

For an asset such as the Concept of Operations [Cohen 99], the inspection team should create more scenarios that examine the main narrative than for the glossary or appendices since the reader of the document will spend more time with the narrative.

Operational profiles are used to establish the basis for reliability calculations. The profile of use for an asset may vary significantly from one product to another. Test results concerning the interaction of a complex set of components can be very different if the frequency of use, sequence of use, or timing between invocations of a set of actions changes significantly. This can adversely affect the reliability of the software. Runeson and Regnell describe research into how these varying operational profiles can be handled [Runeson 98].

3.6 Acceptance Testing of Mined and Acquired Assets

The product line organization should conduct an acceptance test for every core asset that is either mined from internal legacy software or acquired from an external vendor. The testing process for assets produced by the product line organization assures a certain level of quality. When assets are acquired from a third party or mined from existing products within the company, that level of quality is not assured. Each external artifact must be tested against the same quality standard before it is accepted as an asset and used to construct products.

The acceptance test set is developed partly from existing tests and partly from tests created specifically for this purpose. A mined asset may be an implementation of an interface for which functional test cases have already been created. If the mined asset is a conceptually complete module, there should be structural tests as well as functional tests already in the test infrastructure of the project that created the module.

If the product line organization undertakes a large-scale acquisition of assets, one criterion for evaluating bids should be the specificity of the test information to be provided by the vendor. A second criterion is the level of test coverage the vendor certifies to have used when manufacturing the assets.

3.7 Roles and Responsibilities

A set of roles and their associated responsibilities are provided in Table 6.

Table 6:   Roles and Responsibilities for Asset Testing

Generic Role	Asset Test Role
Test Architect	• Identifies families of components that can be tested in the same basic way, such as graphical user interface (GUI) components and hardware drivers • Specifies the structure of test harnesses used with each family
Tester	• Selects and constructs test cases for each asset • Executes tests as soon as an asset is determined to be ready for testing • May also be the developer of the asset
Test Manager	• Determines the test priorities among assets • Assigns resources based on domain expertise
Specifier	• The product line architect will define the interfaces in the architecture that are, in turn, the specifications for the top-level components. Asset developers will also specify additional lower-level components.

3.8 Summary

Testing each individual asset as it is created is cost effective and helps assure its quality. The development process produces specifications, which are inspected to assure conformance to the requirements and the architecture. The details and implementations of that specification are analyzed to create test cases. The analysis of a specification and the test cases derived from it are used to test all implementations of that specification. This specification-based testing must be more extensive in a product line organization to assure that the asset operates correctly across the wide range of contexts into which it will be placed.

Artifacts are tested either manually or automatically. In either case, the test cases constructed from analyzing the specification and implementations are used to guide the testing process. Artifacts that pass a sufficient percentage of the test cases become core assets.

In addition to original development, assets may be found in existing products or purchased. This requires fewer resources than "from-scratch" development. However, this type of asset often requires more testing resources than assets created by the product line organization do.

4 Testing Products

Testing a product requires a different perspective from testing individual assets. The focus shifts from correctness and completeness to conformance. Each product must meet external requirements. The focus in this section is on the testing that determines whether the completed product meets the requirements of its customers.

A product development team's test responsibility also encompasses testing any core assets that are modified specifically for its product. The core-asset developers will have done the majority of asset testing and created a repository of test cases for each asset. Specific tests for the modified assets are addressed in Section 3.

Testing the architecture of any system is, in many ways, the same as testing the completed product. The test scenarios used for architecture evaluation (see Section 3.2.2) are one source of scenarios for building product test cases. Evaluating the architecture is testing for the potential of realizing certain product qualities, while dynamic testing of the completed product is testing for the actual achievement of these qualities.

4.1 Test-Case Derivation

The process for selecting and constructing product-level tests begins as soon as the requirements process for the product line begins. Figure 7 illustrates the hierarchical relationships between artifacts that lead to test cases. Each use case created in the requirements process captures one or more requirements that a specific product must satisfy. The use-case description contains scenarios that describe the "usual course," alternative courses, extensions, and exceptional courses. Each of these scenarios is specialized[3] to a set of test scenarios that contain the additional detail needed to operate the asset during a Guided Inspection [Wiegers 97]. This is done in preparation for the asset-test phase when the various analysis and design models are inspected, as described in Section 3.2. Later these test scenarios are further specialized to construct the executable test cases to be used during dynamic system testing.

---

[3] The specialization is done by giving each unknown value in the scenario a more specific value. This value will be sufficient for the next level of testing. Each unknown can be given multiple values so specialization produces multiple outputs from a single input.
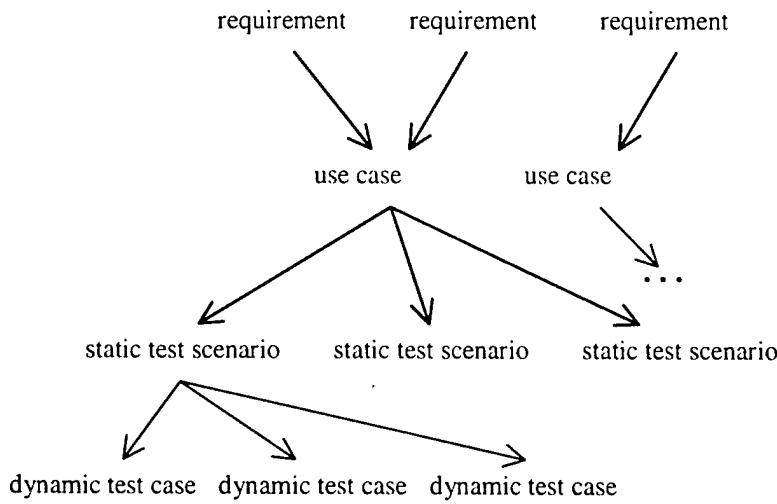
Figure 7: *Deriving Test Cases*

For example, the usual-case scenario from a use case in the wireless product line shown in Table 7 could have the following pre-conditions and result in the test scenario shown in Table 8:

- The device has room left to store one additional entry in the address book.

- The device is not currently using any communication channels.

Table 7: *"Add AddressBook Entry" Use-Case Success Scenario*

When the user does this:	The system responds by doing this:
Selects the "Add entry to address book" function	1. Initializing a blank entry 2. Displaying the blank entry on the screen
Enters the requested information	1. Storing the information in flash memory 2. Displaying confirmation to the user

Table 8: *Test Scenario for "Add AddressBook Entry" Use Case*

When the user does this:	The system responds by doing this:
Selects the "Add entry to address book" function	1. Initializing a blank entry 2. Displaying the blank entry on the screen
Enters: John D. McGregor Software Engineering Institute Pittsburgh, PA 412-555-1212	1. Storing the information in flash memory 2. Displaying the message "Entry added to address book"

In a product line organization, the above use case will be specialized to other use cases for specific products, as shown in Figure 8.
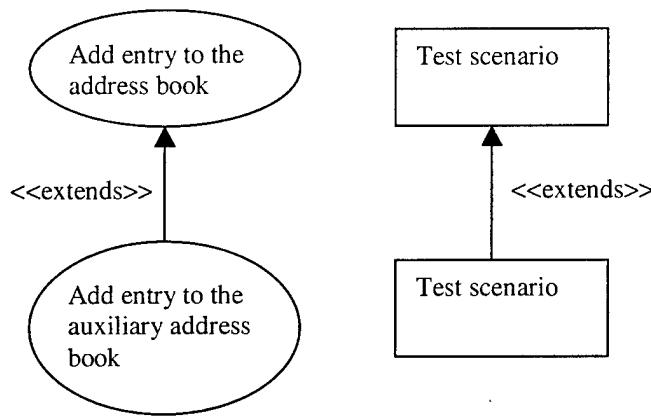

Figure 8:   Form Follows Form

For example, the specialized use case shown in Table 9 could have the following pre-conditions and result in the test scenario shown in Table 10:

• The device has auxiliary memory installed

• The device has room left to store one additional entry in the address book

• The device is not currently using any communication channels

Table 9:   Specialized Use Case

When the user does this:	The system responds by doing this:
Selects the "Add entry to auxiliary address book" function	1. Initializing a blank entry 2. Displaying the blank entry on the screen
Enters the requested information	1. Storing the information in flash memory 2. Displaying confirmation to the user

Table 10: Test Scenario for Specialized Use Case

When the user does this:	The system responds by doing this:
1. Selects auxiliary memory functions 2. Selects the "Add entry to address book" function	1. Initializing a blank entry 2. Displaying the blank entry on the screen
Enters: John D. McGregor Software Engineering Institute Pittsburgh, PA 412-555-1212	1. Storing the information in flash memory 2. Displaying the message "Entry added to address book"

4.2 Test Suite Design

The large number of variation points and possible configurations of products in the product line result in a large number of possible tests. This large number of tests is handled in two ways:

- The analysis method determines the number of test cases for each feature based on priorities of the requirements.

- The test design method configures test sets to have the minimum number of test cases that will provide acceptable effectiveness.

4.2.1 Analysis

The typical use-case template includes a field for representing the relative frequency of the use case among all the other use cases. The core-asset developers initially set these frequencies and then the product developers adjust the frequencies to reflect the exact usage of their product. When the test set is based on these frequencies, testing provides the data necessary to compute reliability of the product. Table 11 describes the priorities of use cases.

Table 11: Priorities of Use Cases

Use-Case Scenario	Frequency
Make a telephone call.	Medium
Connect to the Internet.	High
Look up an address in the address book.	Low

This frequency information provides a form of operational profile for the product being developed. Test sets are structured so that the most frequently used, or highest priority, use cases provide the largest number of test cases. By testing the product in the same way that it will be used, the test results provide a basis for computing a reliability measure for the product [Musa 99].

The tester role performs an analysis by

- ranking the use cases based on frequency
- estimating the number of test cases
  - needed to achieve the desired level of test coverage
  - based on resource limitations
- assigning a number of test cases per use case

The result of this analysis is the specification of the number of test cases needed for each use case to support reliability calculations. The tester must still design tests that provide the desired degree of functional and structural coverage.

4.2.2 Design

Test-case design will begin with the usual course scenario. As the frequency of the use case, and thus the required number of test cases, increases, test-case design will use more of the secondary scenarios. Also, multiple tests are created from each scenario. Very frequent use cases are associated with multiple test cases derived from each scenario.

Designing system tests in a product line is accomplished in two stages. First the product line team creates generic test cases from the use-case scenarios. These test cases contain a parameter for each variation point that is encountered in the scenario. The product team performs the second stage of test design by providing the appropriate parameters, which are still high level. The product-test designer must select the appropriate values to give complete coverage.

Every variation point becomes a factor in a combinatorial test design. Every alternative value for the variation point becomes a level of the variation point factor. In the wireless device example, the type of display contained in the device would be a factor. Each different type of display would be a level in the design. As described in Section 2.5, each level of each factor will be tested with each level of every other factor, but all possible combinations of all levels of all factors will not be tested together.

4.3 Test Composition and Reuse

One technique for representing the variation points in a product line architecture has been proposed by Bachmann and Bass [Bachmann 01]. Figure 9 and Figure 10 show how a variation point might be represented in an architecture. At the interaction between Module A and B and Module A and D, different parameters must be provided by A depending on whether B or D is present. Different answers are produced depending on the configuration.
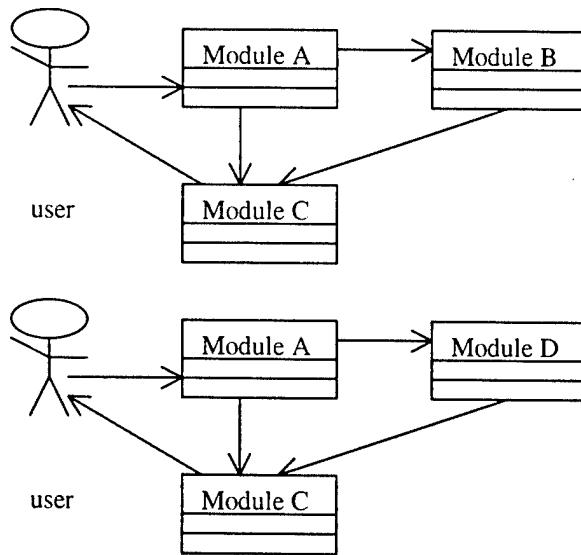
Figure 9: Two Variant Values at a Variation Point
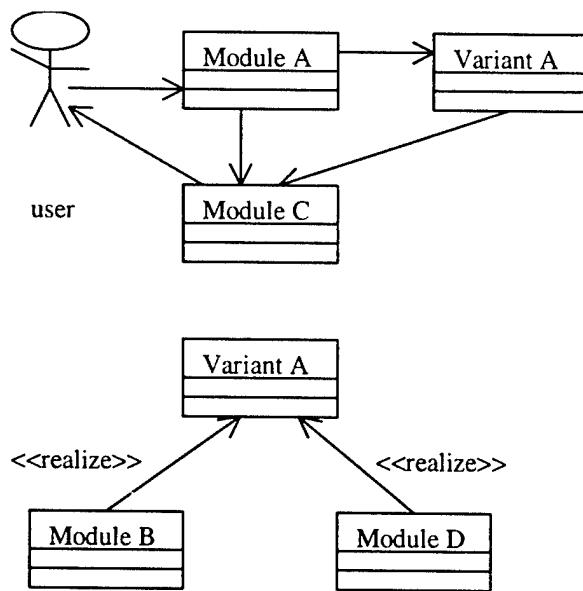
Figure 10: Variant Definition

The test architecture responds to these variation points by composing partial test cases to build a complete test case. Figure 11 shows how the test architecture parallels the architecture of the product line assets. Either Test Module B or Test Module D (depending on the configuration of the product) replaces test Variant A in the test software. The Test Module B or D provides the appropriate parameters to Test Module A so that it can provide the appropriate values to test the interaction between Module A and the chosen variation.
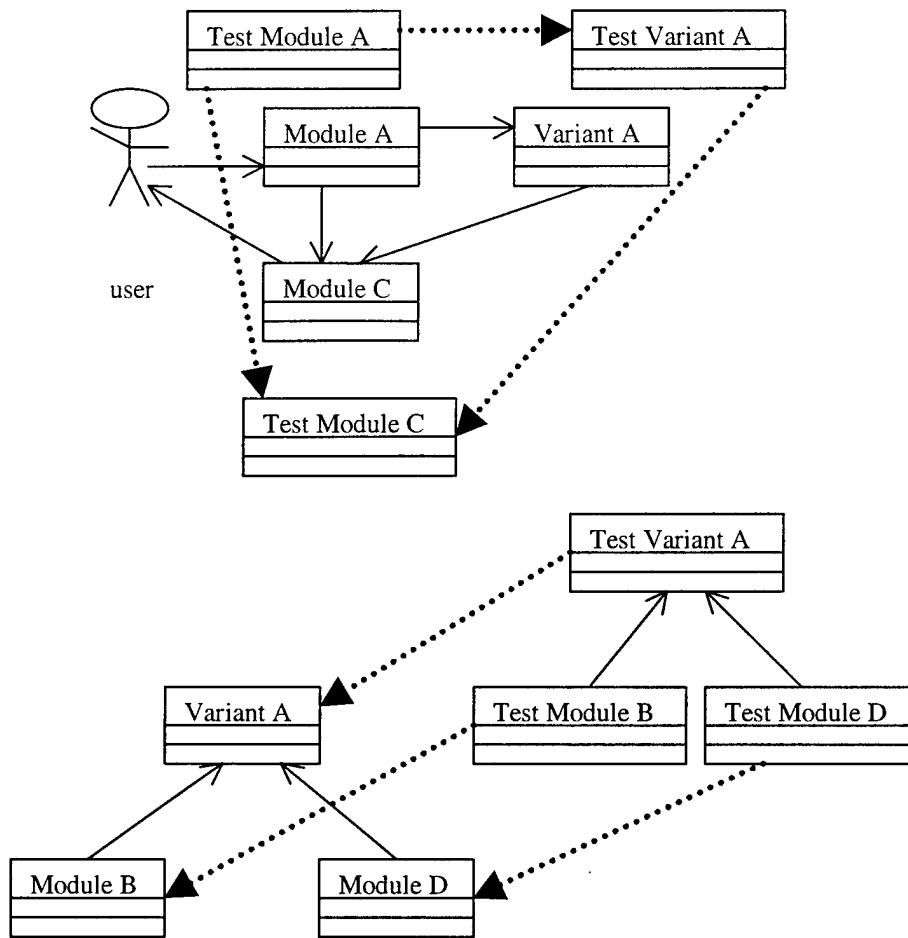
Figure 11:   Composing Partial Test Cases

4.4 Test Construction

According to Bachmann and Bass [Bachmann 01], the techniques described below are the
ways in which products can be realized from a product line description. Each of these leads to
a technique for constructing test software.

- generators - When products are automatically generated from a formal description, the
  same type of description is used to automatically generate the test suite. The formal de-
  scription of the test cases will be compact and easy to maintain.

- configuration management system - When products are assembled based on configura-
  tion descriptions, the configuration description includes test-case definitions. These are
  usually test cases that have already been completely assembled, which decreases the
  maintainability of the test suite.

- compilation - When products are formed based on switches set at compile time, the par-
  tial test cases to be assembled are associated with the same switch as the variant to which
  they are associated. Traceability between the switches and features establishes the same
  association as for the switches and variation points.

- adaptation at start-up - When a product is dynamically configured at system start-up, a separate approach will be needed for the test cases. They are still associated with the variants as described in the previous section, but may need to be compiled separately from a configuration description.

- adaptation during normal execution - When a product is contained in an executable that contains all possible variations or loads them dynamically during execution, a separate approach will be needed for the test cases. They are still associated with the variants as described in the previous section, but may need to be compiled separately from a configuration description.

4.5 Operational Profile for a Product

Testing based on the operational profile is used to compute the reliability of the product. This testing is conducted after the majority of defects have been detected and removed. The test cases used during product creation can be used again for this computation.

As stated in Section 3.5, an operational profile describes the relative frequency of use of certain behaviors. For products, an operational profile indicates the relative frequency with which the features are used. This information has been captured in the use-case model of the requirements during product construction. Most of the popular templates for use cases capture information to be used for this and similar purposes. Cockburn [Cockburn 97] describes a field termed "priority" that can be used as a measure of relative frequency, such as the use-case diagram shown in Figure 12.
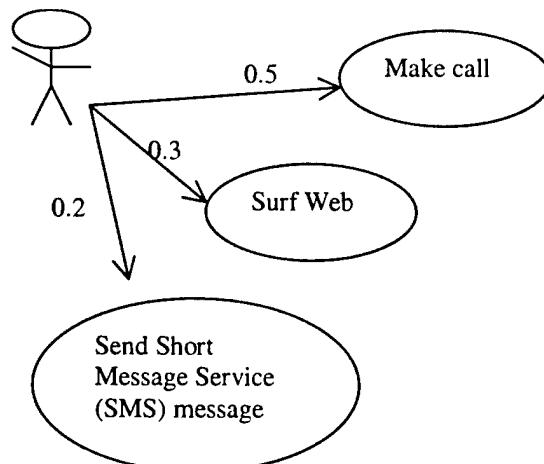

*Figure 12:   Use-Case Model for Wireless Device*
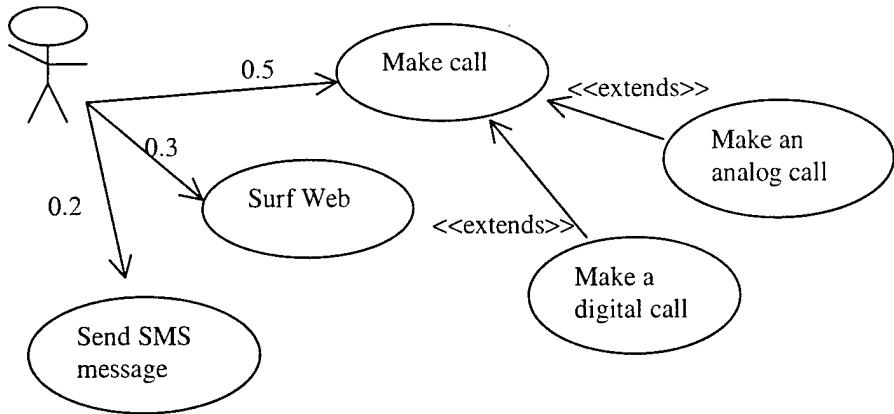
Figure 13:   Variants at the Product Level

One level of variation points for a product is at the use-case level. The use-case model defines an abstract use that is the placeholder for the specific variant chosen for a particular product, as shown in Figure 13. By associating test cases with a use case (as illustrated in Figure 14), a reliability test set for the product can be selected from the test database. This database contains test cases that have been created during the product-creation process.

The variation in patterns of use from one product to another is constrained due to the regions of commonality in the architecture. The product test cases created from use cases form an asset base from which a product team can select a system test set. The test-case asset base contains the contributed test cases from all of the product teams. By selecting from the asset base according to the operational profile for its particular product, a product-test team can create some portion of its test set from the preexisting test cases. The team may need to create additional test cases that cover specific combinations of variations that have not yet been encountered in the product line.

4.6 Testing and System Qualities

There is a direct relationship between the types of testing activities used by an organization and the qualities that are driving the development. Some of these properties can be observed during an execution of the product while other properties can only be observed by a static inspection of the product artifacts. Just as the development team maintains traceability links between requirements and code, the test team needs to maintain traceability between the quality drivers and specific testing techniques.

There are a large number of qualities that can be referred to as architecture drivers [Bass 98]. Table 12 lists a sample of qualities and a brief description of a testing technique.

Table 12: Testing for Qualities

Quality	Testing Method	Product Line Specifics
Reliability	Follow an operational profile to appropriately weigh which features to test most. The percentage of failures is used to compute the reliability.	A product line organization can aggregate test information collected on a feature basis in each product to more accurately estimate the reliability of products that incorporate that feature.
Performance	Construct test scenarios and a test harness that support the accurate measurement of the time required to perform a scenario.	A product line organization can expend sufficient effort to develop performance profiles for components so that the composition of these components will yield the expected performance.
Extensibility	Construct change cases [Ecklund 96] - use cases that are not requirements yet. Use them during the architectural evaluation to examine, in detail, the response of the architecture to the hypothesized extensions.	A product line development process incorporates a certain amount of extensibility as a natural part of the process. The change-case analysis can go beyond the scope of the planned product line.
Security	Construct specific test cases that enter the sections of the system that are supposed to be secure. Attempt each security scenario with both appropriate and inappropriate access privileges.	A product line organization should develop standard security frameworks, and each has an accompanying test framework.

4.7 Roles and Responsibilities

The product team has several responsibilities for testing the completed products, as shown in Table 13. It must test any modified assets for correctness and test the product for conformance to the product's requirements. The product line team has the responsibility of beginning the construction of product test cases. This will provide feedback on the testability of the requirements as well as providing a set of generic test cases that can be specialized by each product team.

Table 13: Roles and Responsibilities in Product Testing

Generic Role	Product-Test Role
Test Architect	Designs a test environment that integrates with the execution environment for the system under development; creates an architecture that is compatible with the types of testing required
Tester	Works at the functional level to create and execute tests that cover the qualities that are driving system development
Test Manager	Determines the resources that will be needed for all of the types of product testing that will be required based on the qualities that are driving system development
Specifier	For products, the specifier is the use-case writer. This role will be present both at the product line and product levels.

4.8 Summary

Testing a product should determine whether the functional and nonfunctional requirements have been met. The core-asset developers provide a generic set of tests based on the requirements for the product line architecture. This includes a set of partial test cases that focus particularly on the specific points of variation between products. These partial test cases are composed to form test cases for a product that has been composed of corresponding variants. The method by which this composition is accomplished varies, but each approach to composition has an associated approach to test-case derivation.

A product is tested for

- conformance to functional requirements
- achievement of each quality attribute

5 Core Assets of Testing

The various testing activities conducted during the development of components and products create a number of artifacts, some of which are core assets. These artifacts, which were defined in Section 2.1, include

- test plans
- test cases
- test reports
- test data sets
- test software and scripts

Test artifacts are managed in much the same way that production artifacts are:

- Test cases, data sets, software and scripts are version controlled.
- All test artifacts are under the control of the configuration management system so that when a specific build of a system is recreated, the appropriate test artifacts are also available.

5.1 Qualities of Test Software

The construction of test assets is guided by the quality attributes described below.

5.1.1 Traceability

Test software must be easily associated with the production software that it is designed to test and with the requirements for that production software. This can be achieved in two basic ways: physically and logically.

5.1.1.1 Physical Traceability

The testing assets are tagged internally with information that associates them with specific core or product assets. For example, the code for a test case can include a reference to the module that it is intended to test. This approach lacks flexibility and inhibits reuse.

5.1.1.2 Logical Traceability

The testing assets are stored at a location that is associated with where the asset is stored. The configuration management system can define an association between the test modules and the core or product software modules.

5.1.2 Modifiability

The test software must be easy to change in order to keep pace with the many changes to the production software that will occur in an iterative, incremental development process. This can be achieved through an appropriate software architecture for the test software and an expressive implementation language.

5.1.2.1 Test Architecture

Modules in the test architecture correspond to fundamental types of test artifacts. The concept of a test case is an identifiable unit in the test architecture. This results in test cases that are modular and easily changeable. The test architecture details the structures that enhance traceability such as the associations described in Section 5.1.1.

5.1.2.2 Implementation Language

Test software must be built quickly but also accurately. A language that supports type checking will eliminate many errors that might go undetected in an inspection. Using the same programming language as the development teams allows the testers to commission the test software from the developers and takes advantage of the programming tools already available.

5.1.3 Configurability

The test software must be amenable to being configured to handle as many variations of the software under test as possible. Test harnesses will be used at both the product line and product levels. The environments at these two levels will be different, and the test software should be usable at both levels.

5.2 Test Automation

Automating testing tasks is necessary in a product line organization because there will be multiple versions of the tests that will be reapplied in multiple iterations on a single product and across multiple products. Many of the tools used by the development organization will also be used for testing activities. The "Tool Support" practice area in the Framework for Software Product Line Practice developed by the SEI provides a comprehensive discussion [Clements 02b]. In particular, a configuration management tool will be used to manage the multiple versions of each test. This tool or a more specialized one is used to provide traceability between the tests, the requirements or specification that they verify and the artifacts to which they are applied.

The benefits of automation are increased when the following basic principles are followed:

- Each unit of automation should correspond to a natural unit in the development process for product lines: for example, group test cases and other assets according to the component to which they apply.

- Each unit of automation should be sufficiently abstract to be robust in the face of minor changes. For example, automating at the interface level means that a test set can still be applied even when the implementation behind that interface changes.

- Each unit of automation should be designed to be composed of other units. The values returned by a piece of code or the exceptions thrown should follow an overall design standard. For example, have the test-case code return a value of true or false depending on whether the test case passed or failed. Then an aggregating test case can decide whether to proceed with the next step in a larger test case or to abort since a failure has already occurred.

Test-case creation is one of the most time-consuming operations in software development. This effort can be reduced by starting at the product line level, creating general test cases, and then specializing them to a specific product; however, the effort is still considerable. Applying these test cases to the artifact can also be a major resource drain. The automation of test-case creation and execution is accomplished in one of several ways: custom test harnesses, testing tools with scripting languages, or test-case generators.

5.2.1 Custom Test Harnesses

Test environments such as Junit [Junit] provide an execution environment for test cases that have been constructed using a basic programming language. These harnesses are useful at the unit-test level or as a simulation environment for embedded systems. It is easy to create suites of test cases that are structured like the architecture of the software, as discussed in Section 2.3.

This approach takes more initial effort per test case than the other techniques. The tradeoff is increased flexibility and compatibility. Object-oriented programming languages can be used to create flexible environments that easily adapt to changes in the specifications and implementations. Custom programming makes it possible to develop tests that are compatible with other tools being used in the development process.

5.2.2 Test Scripts

Test tools provide an environment for executing test cases and a language for constructing test cases. The languages are becoming more sophisticated so that the line between custom-programming environments, such as Junit, and commercial test tools is blurring. Scripts can be written at a general level and then specialized using inheritance. These tools provide a means of executing scripts, and a few also provide a repository in which test results can be stored and compared from one round of testing to the next.

For the end-user-level testing of systems with GUIs, test scripts can be generated automatically using a "capture and playback" mode of a test tool so that, once a human tester creates a test case, it can be repeated without human intervention. These scripts use the names of objects in the interface and are robust with respect to their rearrangements.

These tools are perhaps the easiest to use and only require human resources for the initial round of testing. However, they do not provide any guidance on the semantics of which tests to create or what the tests contain.

5.2.3 Test-Case Generators

Generators use templates to generate test cases with a fixed format and limited variability. These generators require a large amount of effort initially because they require a formal representation of the specifications that are being tested. Creating the templates also requires effort, which is amortized over the products that use similar types of assets. It is further amortized when the development approach also uses a generator approach. Weiss and associates [Ardis 00] have created languages that are specific to product lines and that are used to define products. This same language is the basis for the test-case generator.

Generators do not include an environment for the execution of test cases; however, they do provide more guidance about the content of individual test cases. Generators can lead to more complete testing because as each new test case is generated, different values for inputs can be selected. This results in more complete coverage of the input space.

5.3 Organizing Test Assets

The test assets are organized around requirements and specifications, as explained below.

5.3.1 Requirements

Product test cases are based on the use cases that comprise the product's requirements model. A use-case model defines a set of relationships between use cases that structure the model to reduce redundancy and simplify maintenance. Test cases are decomposed and aggregated in the same ways in which the use cases are structured.

Consider Figure 14. It shows a use-case structure on the left-hand side and the accompanying test-case structure on the right-hand side.
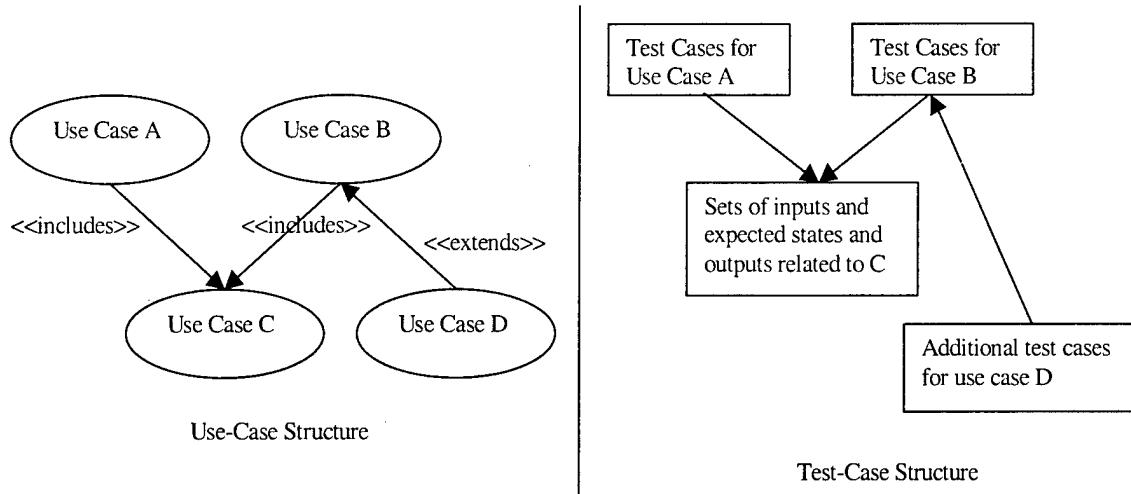
Figure 14:   Use Cases and Test Cases

5.3.2 Specifications

Test cases used on code assets are based on specifications. The specifications are related to specific interfaces that in turn are organized based on the structure of the architecture. The representations in the architecture that identify variation points provide the opportunity to structure alternative inputs, state checks, and outputs for test cases.

5.4 Turning Artifacts into Core Assets

In order for the many test artifacts to be core assets, the artifacts must be defined and structured in ways that make them reusable and modifiable. Techniques for both non-executable and executable artifacts are presented here.

5.4.1 Non-Executable Artifacts

Documents such as test plans will be of value if they capture useful information and help organize the information so that relationships become more obvious. The construction of these documents is not a goal of the project; however if defined properly, plans, designs, and reports can reduce the effort needed to perform certain construction tasks.

5.4.1.1 Standardization

Test artifacts should be based on published standards such as those of the Institute of Electrical and Electronics Engineers (IEEE). Standards are created from the input of many people from many domains based on much experience. By starting with a standard, an artifact may be more comprehensive than one that is based on local experience alone.

5.4.1.2 Customization

The standards-based artifacts should be customized to the product line. Standards are by nature inclusive. There are often items that do not apply to the local situation and can be eliminated. The artifact templates are version controlled so that experience can guide the evolution of these artifacts.

5.4.2 Executable Artifacts

Test cases are assets if they are designed in layers and are closely related to the code that they are intended to test.

5.4.2.1 Abstraction

The test software should be designed in layers of abstraction. This provides the ability for test designers to select the appropriate level of detail upon which they will base their artifacts. The two obvious levels of abstraction correspond to the product line/product structure of the organization. Within each of these two levels, there are opportunities for further levels based on incremental definitions.

5.4.2.2 Association

Each test artifact must be associated with one or more product line asset. Section 5.1.1 illustrates one association between test modules and production modules.

5.5 Testing the Tests

As with any part of the product development process, there must be validation activities that assure the quality of the test artifacts. Table 14 provides examples of testing some of the non-executable artifacts. The executable assets will also be inspected; however, the other validation factor is the comparison of the expected result in a test case to the actual result. When a failure is indicated during the initial use of test cases, both the test case and the software under test are suspect.

Table 14: Testing Test Assets

Asset	Test Technique	Coverage Measure
Test plan	• Syntactic audit by test-process team • Inspection by domain experts	1. Periodic audit by test-process team of every test plan to assure conformance with organizational standards Audit to assure completeness of test-case specification
Test cases	• Syntactic audit by test-process team • Inspection by domain experts	1. Periodic audit by test-process team of a percentage of test cases for format 2. Periodic audit by test-process team of a percentage of test cases for format and an inspection of each test case for accuracy by domain experts
Test reports	• Syntactic audit by test-process team • Inspection by domain experts	1. Periodic audit by test process team of a percentage of test reports for format

5.6 Summary

Test assets are important and expensive resources of a product line organization. These assets should be as compatible as possible with the tools used by the development organization so that policies and procedures can be uniformly applied to documents and code whether they are development artifacts or testing artifacts.

6 Summary and Future Work

This report has presented a comprehensive set of testing techniques in a product line organization. These techniques complement the development activities in the development process for product lines. These development activities span a range from performing analyses and creating designs to writing and integrating program modules. The testing techniques correspondingly span both static and dynamic testing actions. Those techniques are similar to those used in any software development effort but are unique in degree.

Actions can be taken at several levels to maximize the benefits of the product line testing process:

- The test software architecture must be closely related to the software architecture of the product line. Testers on multiple product teams will use the tests. Having the close association between architectures improves the tester's ability to quickly identify which tests to use.

- The test software and other test artifacts must be more completely factored. New, often unanticipated, products are composed from product line assets. Partial test cases must be composed to build complete test cases for each of these products.

- Test plans must take more advantage of statistically based test-selection techniques. The large number of variations defined in the product line architecture lead to a very large number of potential products to test. Test designers apply statistical techniques to reduce the number of test cases needed to provide adequate test coverage.

In recent years testing has gained more visibility in the development process as quality issues have risen in priority for software development organizations. As these organizations adopt a product line organization approach, there are many opportunities to improve the testing process and to reduce the amount of resources required for testing. These opportunities include

- architecture-driven testing—A product line architecture defines component specifications and identifies the qualities that will drive all aspects of design. These factors make the architecture a prime source of test-design information.

- reuse of test assets—Many of the benefits of a product line center around reuse and testing is no different. Test artifacts can be developed around a test architecture that closely parallels the product line architecture. This improves the maintainability of the test assets by providing a means of tracing the changes from the code produced by core-asset builders and product builders into the test code.

- early identification of defects—Inspection techniques can be improved by using a testing perspective through the development of scenarios that guide the inspection. This tech-

nique provides testers and developers with a technique that ensures that the model or document being inspected is systematically, objectively, and thoroughly covered.

Product line practice is rapidly evolving. Additional work needs to be done to improve the test practice to take full advantage of the artifacts produced by the product line activities. Some examples of directions for further work include

- Develop improved specification notations to make the information more usable.

- Develop more effective sampling techniques for selecting regression-test sets from the original functional, structural, and interaction tests.

- Develop more comprehensive techniques for identifying interaction defects among program variants.

References

[Ardis 00]   Ardis, Mark; Daley, Nigel; Hoffman, Daniel; & Weiss, David. "Software Product Lines: a Case Study." *Software Practice and Experience 30*, 7 (June 2000): 825 - 847.

[Bachmann 01]   Bachmann, Felix & Bass, Len. "Managing Variability in Software Architectures," 126-132 *Proceedings of the Symposium on Software Reusability*. Toronto, Canada, May 18-20, 2001. New York, NY: Association for Computing Machinery, 2001.

[Bass 98]   Bass, Len; Clements, Paul; & Kazman, Rick. *Software Architecture in Practice*. Reading, MA: Addison-Wesley, 1998.

[Beizer 90]   Beizer, Boris. *Software Testing Techniques*. Boston, MA: International Thomson Computer Press, 1990.

[Chow 78]   Chow, Tsun. "Testing Software Designs Modeled by Finite-State Machines." *Transactions on Software Engineering SE-4*, 3 (May 1978): 178-187.

[Clements 02a]   Clements, Paul; Kazman, Rick; & Klein, Mark. *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA: Addison-Wesley, 2002.

[Clements 02b]   Clements, Paul & Northrop, Linda M. *Software Product Lines: Practices and Patterns*. Boston, MA: Addison-Wesley, 2002.

[Cockburn 97]   Cockburn, Alistair. "Goals and Use Cases." *Journal of Object-Oriented Programming 10*, 5 (September 1997): 35–40.

[Cohen 96]   Cohen, David M.; Dalal, Siddhartha R.; Parelius, Jesse; & Patton, Gardner C. "The Combinatorial Design Approach to Automatic Test Generation." *IEEE Software 13*, 5 (September 1996): 83-88.

[Cohen 99]   Cohen. Sholom. *Guidelines for Developing a Product Line Concept of Operations* (CMU/SEI-99-TR-008. ADA367714). Pittsburgh. PA: Software Engineering Institute, Carnegie Mellon University, 1999. Available at <http:www.sei.cmu.edu/publications/documents/99.reports/99tr008/99tr008abstract.html>.

[Ebenau 94]   Ebenau, Robert G. & Strauss, Susan H. *Software Inspection Process*. New York, NY: McGraw-Hill, 1994.

[Ecklund 96]   Ecklund, Earl F.; & Delcambre, Lois M. L. "Change Cases: Use Cases that identify Future Requirements," 342-358. *Proceedings of OOPSLA '96*. San Jose, CA, October 6-10, 1996. New York, NY: Association for Computing Machinery, 1996.

[Fagan 76]   Fagan, M. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal 15*, 3 (1976): 182-211.

[Griss 98]   Griss, Martin L.; Favaro, John; & d'Alessandro, Massimo. "Integrating Feature Modeling with the RSEB," 76-85. *Proceedings of the Fifth International Conference on Software Reuse*. Victoria, British Columbia, Canada, June 2-5, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998.

[IBM 97]   IBM Object-Oriented Technology Center. *Developing Object-Oriented Software: An Experience-Based Approach*. Upper Saddle River, NJ: Prentice Hall, 1997.

[IEEE 90]   Institute of Electrical and Electronics Engineers. IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.121990. New York, NY: IEEE, 1990.

[Jacobson 97]   Jacobson, Ivar; Griss, Martin; & Jonsson, Patrik. *Software Reuse: Architecture, Process, and Organization for Business Success*. New York, NY: Addison Wesley Longman, 1997.

[Junit]   Junit. <http://junit.sourceforge.net/>.

[Kang 90]   Kang, Kyo C.; Cohen, Sholom G.; Hess, James A.; Novak, William E.; & Peterson, A. Spencer. *Feature-Oriented Domain Analysis Feasibility Study* (CMU/SEI-90-TR-21, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990. Available at <http://www.sei.cmu.edu/publications/documents/90.reports/90.tr.021.html>.

[McGregor 00] McGregor, John D. "A Testing Perspective." *Journal of Software Testing Professionals 1,* 4 (December 2000): 11-15.

[McGregor 01] McGregor, John D. & Sykes, David A. *A Practical Guide to Testing Object-Oriented Software.* Boston, MA: Addison-Wesley, 2001.

[Meyer 97]   Meyer, Bertrand. *Object-Oriented Software Construction, second edition.* Upper Saddle River, NJ: Prentice Hall, 1997.

[Musa 99]   Musa, John D. *Software Reliability Engineered Testing.* New York, NY: McGraw-Hill, 1999.

[O'Neill 89]   O'Neill, Don. *Software Inspections Course and Lab.* Pittsburgh, PA. Software Engineering Institute, Carnegie Mellon University, 1989.

[O'Neill 96]   O'Neill, Don. "National Software Quality Experiment Results 1992 - 1996." *Proceedings of the Eighth Annual Software Technology Conference.* Salt Lake City, UT, April 21-26, 1996. Hill AFB, UT: Software Technology Support Center, 1996.

[Phadke 89]   Phadke, Madhav S. *Quality Engineering Using Robust Design.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[Pleeger 98]   Pfleeger, Shari Lawrence. *Software Engineering: Theory and Practice.* Englewood Cliffs, NJ: Prentice-Hall, 1998.

[Rumbaugh 99] Rumbaugh, James; Jacobson, Ivar; & Booch, Grady. *The Unified Modeling Language Reference Manual.* Reading, MA: Addison-Wesley, 1999.

[Runeson 98]   Runeson, Per & Regnell, Bjorn. "Derivation of an Integrated Operational Profile and Use-case Model," 70-79. *Proceedings 9th International Symposium on Software Reliability Engineering (ISSRE98).* Paderborn Germany, November 4-7, 1998. Los Alamitos, CA: IEEE Computer Society, 1998.

[Thayer 97]   Thayer, Richard H. & Dorfman, Merlin. *Software Requirements Engineering..* Los Alamitos, CA: IEEE Computer Society Press, 1997.

[Warmer 99]   Warmer, Jos B. & Kleppe, G. *The Object Constraint Language: Precise Modeling with UML.* Reading, MA: Addison-Wesley, 1999.

[Weiss 99]   Weiss, David M.; Tau, Chi; & Lai, Robert. *Software Product-Line Engineering.* Reading, MA: Addison-Wesley, 1999.

[Wiegers 97]   Wiegers, Karl E. "Listening to the Customer's Voice." *Software Development 5,* 3 (March 1997): 49-55.

Appendix   Wireless Device Product Line

Products are currently being manufactured and marketed that blend the features and characteristics of several electronic devices. Cellular telephones are being expanded to include the features of personal digital assistants (PDA), Internet appliances, and MP3 devices. Manufacturers are creating product lines in which the products have different combinations of features and different price ranges. In addition, telephones are being created that handle different communication protocols.
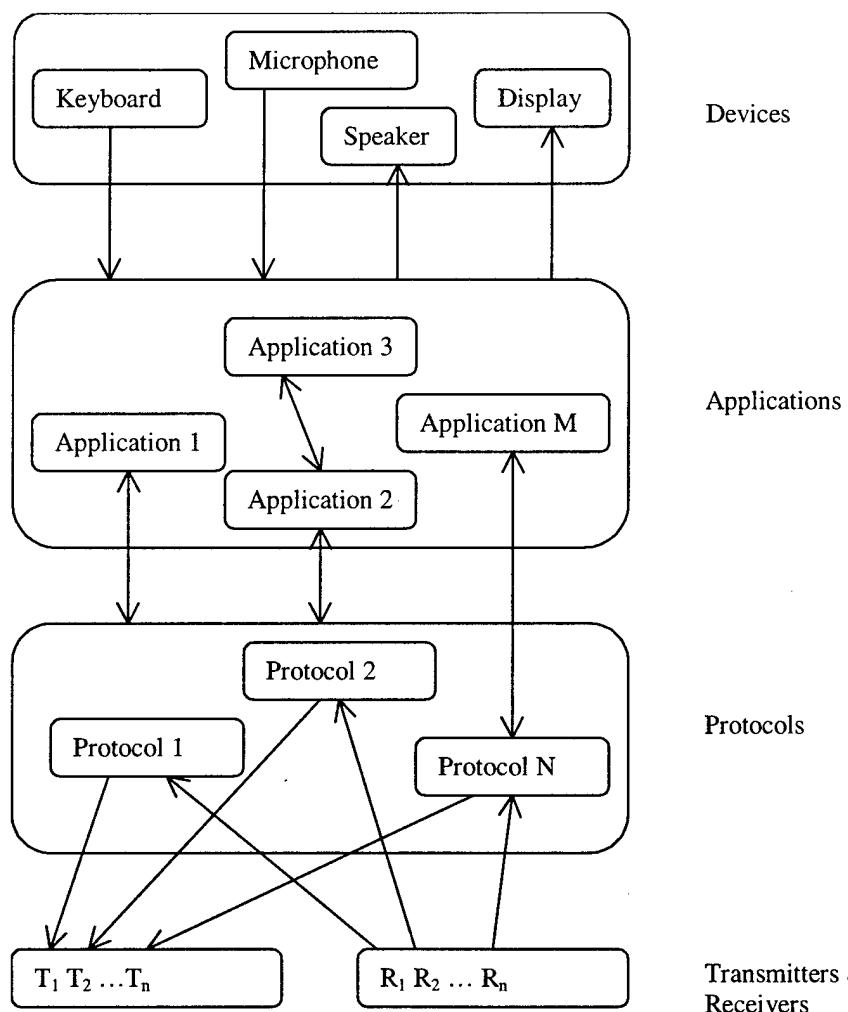
Figure 15:   Wireless Device Architecture

The dependency view of the architecture shown in Figure 15 provides an overview of a basic wireless device. The boxes in the diagram represent conceptual modules. and the arrows indicate information flow. A signal is picked up by a receiver (R) and then decoded by a protocol state machine. The interpreted signal is used as input to applications such as the cell telephone or Web browser. These applications can transmit information back using the same protocol, or they can use a local device such as the display to interact with the user. Other applications, such as calendar and address books, may not use the wireless portion of the device.

The deployment view shown in Figure 16 illustrates the deployment of four processes. The smaller node in the diagram is a special-purpose accelerator chip.
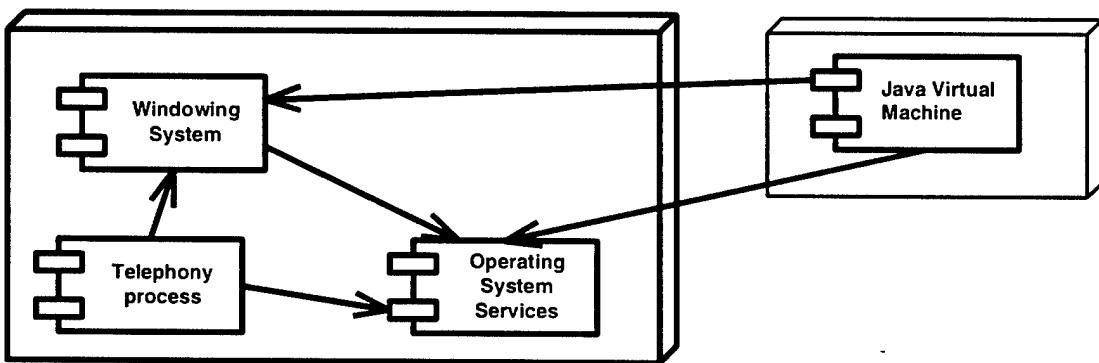

Figure 16: Main Processor Plus Accelerator

These wireless devices have planned interactions and ad hoc interactions. The address book is a stand-alone application that can be edited and searched. A phone call can be made directly from a "page" in the address book. These interactions are part of the REQUIRES clause of the specification of each component. Ad hoc interactions occur when two applications try to use a device at the same time. For example, the alarm clock and the incoming call indicator both attempt to use the speaker at the same time.

The deployment view also groups together modules that reside in the same process. One such group is the set of applications that run in the windowing system. The deployment view shows a standard processor that is augmented by a Java accelerator. The accelerator is used because industry is moving to Java-based applications.

The market for these devices is changing rapidly so *time to market* is a high-priority architecture driver. These devices are used for life-critical issues such as making emergency telephone calls. Thus *reliability* is another important architecture driver.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE December 2001	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Testing a Software Product Line	5. FUNDING NUMBERS F19628-00-C-0003

6. AUTHOR(S)

John D. McGregor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2001-TR-022

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2001-022

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS	12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

A suitably organized and executed test process can contribute to the success of a product line organization. Testing is used to identify defects during construction and to assure that completed products possess the qualities specified for the products. Test-related activities are organized into a test process that is designed to take advantage of the economies of scope and scale that are present in a product line organization. These activities are sequenced and scheduled so that a test activity occurs immediately following the construction activity whose output the test is intended to validate. This report expands on the testing practice area described by Clements and Northrop [Clements 02b]. Test-related activities that can be used to form the test process for a product line organization are described. Product line organizations face unique challenges in testing. This report describes techniques and activities for meeting those challenges.

14. SUBJECT TERMS testing, product lines, software product lines, system testing	15. NUMBER OF PAGES 82

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL